National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

An Automatable Specification Directed Software Testing Method

BY

W. Donald Lawrynuik

A thesis

submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Computing Science

Edmonton, Alberta
Spring 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66741-9

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: W. Donald Lawrynuik

TITLE OF THESIS: An Automatable Specification Directed Software Testing Method

DEGREE : Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Don Lawrynuik

Permanent Address:
Apt # 702 Phase II
2267 Lakeshore Blvd. W.
Toronto Ont.
CANADA

Date: Apr. 24/1991

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "An Automatable Specification Directed Software Testing Method" submitted by W. Donald Lawrynuik in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. M Green ..... Co-Supervisor

Dr. L. White ..... Co-Supervisor

Dr. J. Laski .....

Dr. P. Sorenson .....

Dr. J. You .....

Dr. A. Robinson .....

Date

ABSTRACT

Specification directed testing is an important and largely unexplored area of software testing. We have built a prototype of a system to do specification directed testing of abstract data types. That system is called T-3. In building and using T-3 we have discovered some basic problems that must be addressed before such a system can be useful for most practical applications. This thesis describes those problems and our solutions to them.

A correct software implementation can be viewed as one that (a) does everything it should; and (b) only does what it should. We show that specification directed testing should only be used to investigate (a). A new precise goal for specification directed testing is developed based on this result is presented and developed.

Previous methods for using PROLOG to produce test cases do not work for inequality predicates and cannot test all arbitrary sequences of function calls. A new method for using PROLOG to generate test cases that overcomes these limitations is presented.

Previous methods for test set generation for specification directed software testing are based on the syntax of the axioms in the specifications of the software. We demonstrate the limitations of these methods by presenting simple faults that are never (even with infinite testing) detected by those previous methods. We have developed a new model of the computations of an abstract data type so that a new test set generation methodology could be developed.

We have developed a new test set generation methodology that uses the semantics as well as the syntax of the software specification. It is based on: (1) our new computational model, (2) our method for using PROLOG to generate test cases, and (3) our new goal for specification directed software testing. We have validated this new methodology by comparing the number of faults it reveals to the number of faults revealed by previous methods.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. L.J. White, for his guidance, support, and infinite patience throughout this research.

I am grateful to my co-supervisor, Dr. M. Green for his help over the last few years. I would also like to acknowledge the members of my examining committee, Dr. P Sorenson, Dr. J You, Dr. A. Robinson, and Dr. J. Laski for their valuable comments.

I would also like to thank Miss I.L. Macdonald and Miss M. MacPherson for providing me the financial and personal computing resources necessary to pursue my studies.

Special thanks must go to Edith Drummond, without whom there would be no graduate program in The Department of Computing Science at the University of Alberta. In addition I would like to thank the Power Plant crew, both old and new, for maintaining a good environment for graduate research.

Thanks to Cathy Olson for the patience and tolerance she has shown in keeping everything in perspective and keeping my feet on the ground.

Finally, and most importantly, thanks to my parents, Helen Lawrynuik and Walter Lawrynuik. This thesis is really for them.

TABLE OF CONTENTS

1 INTRODUCTION 1
1.1 The New Testing Methodology 2
1.2 Thesis Outline 6
2 BACKGROUND 10
2.1 Software Testing 10
2.1.1 Specification Directed Testing 12
2.1.2 Summary of the Testing Process 20
2.2 Abstract Data Types 20
2.2.1 Implementation of Abstract Data Types 27
2.3 Test Case Generation Using PROLOG 29
2.3.1 The Test Procedure 29
2.3.2 Using PROLOG 30
2.3.3 Sample Run 33
2.3.4 Partitioning of Instance Classes 35
2.3.5 Problems 38
3 T-3: A PROTOTYPE 40
3.1 Test Harnesses 41
3.1.1 What Does DAISTS Do? 41
3.1.2 How Does DAISTS Work? 42
3.1.3 An Example 45
3.1.4 Specifications 48
3.1.5 DAISTS within the Testing Process 48
3.1.6 Advantages 49
3.1.7 Constraints 50
3.1.8 Experiences 53
3.2 A New Method for Testing ADTs 54
3.2.1 Specification Format 56
3.3 61
3.3.1 Improvements 65

3.4 Problems 66
4 A THEORY FOR ABSTRACT DATA TYPE TESTING 68
4.1 What is Being Tested 68
4.2 Abstract Machine 69
4.3 Goals of the Testing Method 71
4.4 A General Testing Method 76
4.5 The Testing Method 78
4.5.1 Previous Methods 80
4.5.2 Our Method 88
5 A FOUNDATION FOR AUTOMATABLE SPECIFICATION-DIRECTED TESTING METHODS 90
5.1 Software Testing Assumptions 91
5.2 Abstract Data Type Assumptions 93
5.3 Algebraic Specifications 96
5.3.1 Existence 97
5.3.2 Correctness 98
5.3.3 Consistent 99
5.3.4 Sufficiently Complete 100
5.3.5 Other Specification Techniques 102
5.4 Summary 105
6 A NEW METHOD FOR TEST CASE GENERATION USING PROLOG 107
6.1 Instance Classes and Sub-Instance Classes 108
6.2 Testing a Sub-Instance Class 114
6.3 An Updated Test Case Generation Method Using PROLOG 116
6.3.1 120
6.3.2 A More General Constraint Handling Method. 122
6.3.3 The Problem of More General Traces 126
6.3.4 Comparison to Previous Work 130
6.4 An Extended Example 133
6.5 Conclusion 137
7 A COMPUTATIONAL MODEL FOR ABSTRACT DATA TYPES 139
7.1 A Canonical Form for ADTs 140

7.2 Equivalence Spaces 142
7.3 Computation Space 145
7.4 Implications for ADT Testing 151
8 TEST SET GENERATION 153
8.1 Goals of the Testing Method 153
8.2 A General Method For Testing Abstract Data Types 154
8.2.1 Instance Classes 155
8.2.1.1 Possible Failure Set (PFS) 155
8.2.1.2 Testing an Instance class 160
8.2.2 163
8.2.3 Calculating the Relative Benefit of Testing an Instance Class 165
8.2.4 An Algorithm For Testing ADTs 174
8.3 Example 174
8.4 Discussion 181
8.5 Validation 183
8.5.1 Example 188
8.5.2 Model 191
8.5.3 Experimental Procedure 192
8.5.4 Results 194
8.5.5 198
9 SUMMARY AND CONCLUSION 200
9.1 Summary 201
9.2 Conclusion 207
9.3 Future Work 207
GLOSSARY 213
REFERENCES 218
APPENDIX I 226
APPENDIX II 229
Input Files 229
Operation 233
Sample Runs 240
APPENDIX III 242
Generation 245

Results 251
APPENDIX IV 254

LIST OF TABLES

Table 6.1	O-Type Functions Applied to Three Instances	113
Table 7.1	Traces, Instance Classes and Canonical Forms	143
Table 8.1	Sub-instance Classes for $o_i \cdot \gamma$	165
Table 8.2	Order of Traces Tested for Type Bag-c	179
Table 8.3	Test Case Generation for Trace #6 in Table 8.2.	181
Table 8.4	Observable Functionality for T_s	190
Table 8.5	Observable Functionality for T_i	190
Table 8.6	Observable Functionality for T_x	191

LIST OF FIGURES

Figure 1.1	Testing Process Diagram	2
Figure 2.1	An Algebraic Specification of a Queue	21
Figure 2.2	PROLOG Specification of a Queue.	32
Figure 2.3	Generating Test Cases and Results	33
Figure 2.4	Extended Generation Example	37
Figure 3.1	The DAISTS System	43
Figure 3.2	The DAISTS Input Stream	44
Figure 3.3	Sample DAISTS Input	46
Figure 3.4	Procedure from an Axiom	47
Figure 3.5	DAISTS Main Test Driver	47
Figure 3.6	DAISTS within the Testing Process	49
Figure 3.7	An Algebraic Specification with a Hidden Function	51
Figure 3.8	T-3 Testing Process	56
Figure 3.9	MODULA-2 Code to Test an Instance	57
Figure 3.10	Sample Specification Syntax	60
Figure 3.11	A New Testing Method	62
Figure 3.12a	Previous Systems	64
Figure 3.12b	T-3 Testing Tool	64
Figure 4.1	General Testing Method	77
Figure 4.2	An Operational Outline of Our Testing Method	77
Figure 4.3	Code with a Fault over Two Functions	85
Figure 6.1	Application of Assumptions	111
Figure 6.2	Algebraic Specification of Type: BAG	118
Figure 6.3	PROLOG Specification of Type: BAG	118
Figure 6.4	PROLOG Queries for Type: BAG	120
Figure 6.5	Failed Goals	127
Figure 6.6	A New Set of Translation Equivalences	129
Figure 6.7	Additional Axioms for New Bag Type	133
Figure 6.8	PROLOG Specification: BAG-n	135

Figure 6.9	Sample PROLOG Run for Bag-n	137
Figure 7.1	Traces of Length 2	147
Figure 7.2	Traces of Length 3	148
Figure 7.3	Traces of Length 4	149
Figure 7.4	Traces of Length 5	151
Figure 8.1	Algorithm to Test an Instance Class	162
Figure 8.2	The Cost of Testing an Instance Class γ	164
Figure 8.3	Two PFS Elements Removed	169
Figure 8.4	Possible PFS Elements	171
Figure 8.5	Algorithm for testing an ADT	175
Figure 8.6	An Algebraic Specification for a Type Bag-c	177
Figure 8.7	A PROLOG Specification for a Type Bag-c	178
Figure 8.8	RELAY Model of Error Detection	185
Figure 8.9	Algebraic Specification of Queue with Has	189
Figure 8.10	Validation Error Model	192
Figure 8.11	Faults vs. # Tests. For STACK	195
Figure 8.12	Faults vs. #Function Calls. For STACK	196
Figure 8.13	Faults vs. #Tests. For QUEUE WITH HAS	196
Figure 8.14	Faults vs. #Function Calls. For QUEUE WITH HAS	197
Figure 8.15	Faults vs. #Tests Run. For BAG.	197
Figure 8.16	Faults vs. #Function Calls. For BAG.	198
Figure A2.1	Operation of T-3	234
Figure A3.1	Algebraic Specification for a Large Example	244
Figure A4.1	Algorithm for Counting PFS Elements	255

1. INTRODUCTION

Until recently, researchers looking at the problem of software testing only had one formal object upon which to base their work. That object was the program itself. Thus much of software testing research has focused on white box testing. In recent years considerable work has gone into developing techniques and methodologies that produce formal requirements and software specifications. This in turn means we have another formal object (the software specification) upon which to base software testing.

The initial hypothesis of our research has been that a complete testing methodology for abstract data types that is based on software specifications could be developed. This thesis describes the results of our investigation of that hypothesis.

Our first step in investigating specification-directed software testing was to design and build a prototype of a new specification-directed software testing system. That system is called T-3. This prototype tests abstract data type implementations and requires as input only the implementation and the specification. Previous researchers [Gannon81, McMullin83, Day85] have proposed and implemented a compiler-based system, called DAISTS, to test abstract data types (ADTs). That system also requires as input the specifications of the abstract data type and the implementation. Additionally, DAISTS requires a set of test cases and test data. In a separate line of research, several authors [Bouge85b, Bouge86, Wild86] have

reported a methodology for translating algebraic specifications of abstract data types into PROLOG horn clauses and then using them to generate test cases.

1.1. The New Testing Methodology

Figure 1.1 contains the testing process diagram as first presented by Bouge [Bouge85a]. The basic problem we are concerned with is: "Is the implementation correct with respect to its specifications?" That problem is broken down into the sub-problems of abstractly representing or modeling the testing problem, characterizing or constructing an abstract battery of tests for that abstract testing problem, and

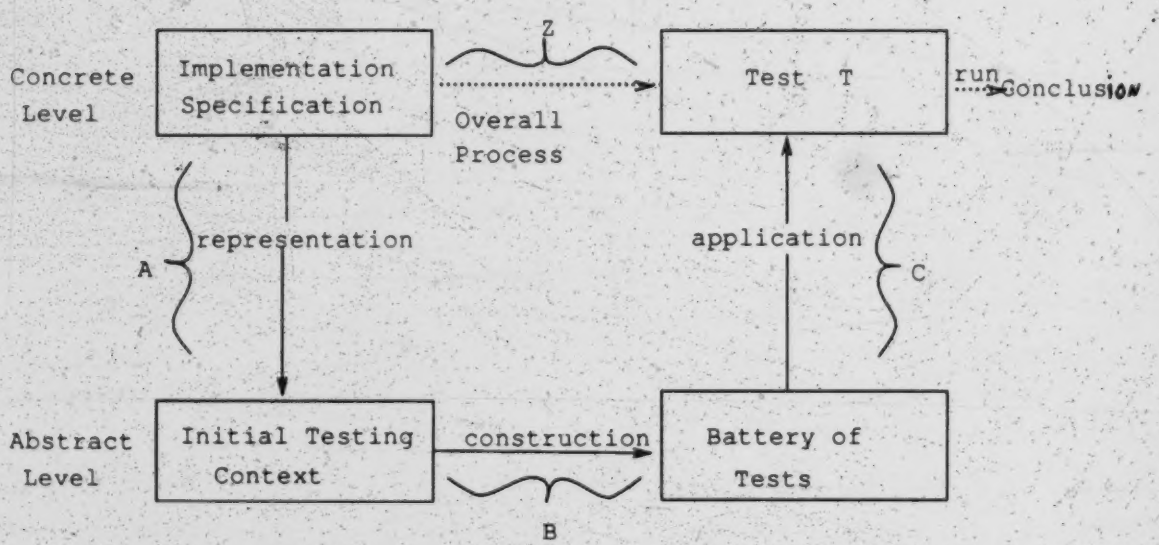


Figure 1.1 : Testing Process Diagram

applying a concrete instantiation of that abstract battery to the the implementation. From the problem to be solved: "Is this implementation correct with respect to its specification?", we build in the representation phase (Step A), "an abstract problem (intuitively) equivalent to the concrete one .... this defines an *initial testing context*" [Bouge85a]. We construct for this initial testing context an acceptable *battery of tests* much as other testers seek an ideal criterion. Having obtained an abstract battery of tests, we can pick some test from it and *apply* (Step C) it to the given problem leading to a conclusion.

In building T-3 we found that the limitations of the overall testing process (Step Z + run) were not the intersection, but rather the union of the limitations of the methods used for representation, construction, and application. We found that an overall testing methodology based on the best current methods (due to the Gannon and Bouge groups) for representation, construction, and application was so limited that it was of no use. The limitations included:

(1) Requiring all functions *used* in the specification to be included in the implementation. It is often useful to use extra functions that will not be externally visible when specifying an ADT. The example in Appendix III uses several such extra functions. The requirement that these extra functions exist in an implementation restricts the way a programmer can implement the ADT, which violates the principle of information hiding, and effectively eliminates the advantages of using

ADTs.

(2) Requiring the system to be tested to be able to determine if two elements of the ADT are the same. This seriously limits the ADTs that can be tested. For example it makes no sense to say that one bag of integers is the same as another. If two second year computing science students were asked to implement a bag of integers they might not use the exact same data structures, yet both could be correct; therefore it would make no sense to ask if one representation of a bag is equal to another.

(3) Requiring that equality be the only predicate used in the specification of the ADT. Predicates such as $>$, $\geq$, and set membership are not allowed. Thus, functionality based on conditions other than strict integer, floating point, or boolean equality of elements is not allowed.

(4) Requiring the assumption that any faults that do occur in the implementation are localized to one function. That is, the tester cannot test for a fault spread over more than one function. For example a fault that involves setting a flag in one function and using it in another function is assumed not to occur by those previous methods. Clearly this is an unreasonable restriction. These are the sort of faults that an average tester might miss and we would like to detect with an automated system.

(5) Requiring the ADT under test to be simple enough that a non-trivial ordering of test cases is not required and potentially infinite backtracking in the production of a test case cannot occur. We found that in practice this limit on simplicity was reached when an ADT's specification had 12 to 15 axioms. Requiring a specification of no more than a dozen axioms means that ADTs with more than four functions cannot be tested. This eliminates most non-trivial abstract data types.

Our testing methodology is a union of new methods we have developed for steps A, B and C (representation, construction, and application). These new methods were developed to get around significant shortcomings in current methods.

We have developed a testing system prototype, called T-3, which implements a new method for the application of a battery of test sets (Step C). This new method can be applied to implementations that do not necessarily contain all functions mentioned in the ADT's specification. This considerably extends the applicability of our overall methodology.

Previous specification directed testing methods only allowed faults in individual functions, a fault could not be spread over more than one function. We attack this problem by developing a new goal for the general software testing process. This refined goal allows us to develop a new method for representing the testing context (Step A). This new method allows a less constrained hypothesis to be made about the

programmer, which in turn allows for a wider class of faults to be detected.

We begin to address the problems of allowing only strict equality predicates and simple ADTs by presenting a foundation for our specification-directed software testing methods. We will use that foundation to develop a new methodology for constructing test cases. That methodology is currently implemented in PROLOG. We have also developed a new method for generating and ranking test sets for ADTs. Together these methods make a new construction methodology (Step B) that is improved in the following ways:

(1) It does not get into potentially infinite backtracking.

(2) It allows for predicates other than strict equality in the specification of the ADT.

(3) It provides a ranking of test cases so that we may produce a sequence of tests with the more useful tests first.

1.2. Thesis Outline

In Chapter 2 we give some background on software testing and on abstract data types. Both of these concepts are key to this thesis.

In Chapter 3 we describe our initial new software testing methodology and a prototypical system called T-3. T-3 has been briefly described previously in [Lawrynuik87]. This new methodology is the initial result of our work and is a combination and extension of previous results in test harness construction [Gannon81,

McMuilin82, McMullin83, Day85] and the use of PROLOG for test set generation [Bouge85b, Choquet86, Wild86, Bouge86]. This extension required an analysis of the DAISTS method in light of the overall testing process (Fig. 1.1). In implementing this methodology with T-3 we discovered:

(1) there was insufficient theoretical basis to allow the ordering of test cases;

(2) it was difficult to determine what assumptions about the software to be tested and its specification were made in previous papers [Gannon81, McMullin82, McMullin83, Day85, Bouge85b, Choquet86, Wild86, Bouge86].

(3) the PROLOG interpreter would often go into infinite backtracking.

The solutions to these problems form the basis for our new testing methodology.

In Chapter 4 we outline the foundations of our new testing methodology. We present a useful theoretical way of viewing abstract data types. That view allows us to determine how an ADT might fail. We examine the overall goal for software testing and develop a justifiable goal for specification-directed testing of ADTs. Finally, we present a general testing method based on that new goal and on our findings of how an ADT might fail.

In Chapter 5 we outline the assumptions for our new specification-directed software testing methodology. This chapter gives the constraints on the domain of software our methodology can test. This description of what can and cannot be

done is important as it allows us to use that methodology as a basis for future work. The constraints on the domain of testable ADTs are considerably more relaxed than has previously been possible.

In Chapter 6 we show why previously published methods [Wild86, Choquet86, Bouge86] for using PROLOG to generate test cases do not work for most abstract data types. We then present the new methods we have developed to generate test cases for a much wider range of ADTs. These methods have been implemented and are used in the overall testing method described in Chapter 8. A detailed example using these methods is given in Appendix III.

We found that to be able to generate a *useful* test set it was necessary to "rank" the test cases. We found that while it was possible to do this based upon the syntax of the test cases and the syntax of the ADT, such syntactic rankings do not give operationally practical test sequences. We found it necessary to look at *what the ADT was actually computing*. In Chapter 7 we outline the model of abstract data type computation we developed to address this problem. We found this computational model useful for ordering tests.

In Chapter 8 we use the results from Chapters 4 through 7 and the new concept of possible failure set(PFS). It is shown that one can view the possible failure set as containing all the ways an ADT can fail. The concept of the possible failure set is used to determine the benefit of running a particular test. Understanding the benefit of

a test allows us to rank the tests and generate a series of tests. A new ADT test set generation methodology is developed based on that ranking. Appendix III contains the results of applying this new methodology to a large complex ADT.

2. BACKGROUND

In this chapter we will give some background on several topics which are key to this thesis.

2.1. Software Testing

Given an implementation of a program P, working over an input domain D, suppose that for each data element d of D, one is able to decide whether the implementation behaves correctly or not with respect to its specification. P(d) denotes the result of executing P with input $d \in D$. Out(d,P(d)) specifies the output requirements for P, i.e., Out(d,P(d)) is true if and only if P(d) is an acceptable result. For a deterministic[1] implementation Out(d,P(d)) is true if and only if P(d) is the (single) result required by the program's specification. In the non-deterministic[2] case Out(d,P(d)) may be true for several different values of P(d). In this case Out(d,P(d)) is false if and only if P(d) violates the specifications of P. In accordance with previous authors, we use OK(d) as an abbreviation for Out(d,F(d)). The correctness of the implementation with respect to its specification, for both the deterministic and non-deterministic cases, can thus be expressed as $\forall d \in D$ OK(d).

---

[1] Deterministic: Denoting a method, process, etc., the resulting effect of which is entirely determined by the inputs and initial state.

[2] Non-deterministic: A mode of computation in which, at certain points, there is a choice of ways to proceed: the computation may be thought of as choosing arbitrarily between them or as splitting into separate copies and pursuing all choices simultaneously. [Illingworth83]

If the predicate OK is to be evaluated, then a test oracle is needed which can check the correctness of any test output. Test oracles may be human experts, tables of values, algorithms for hand calculations or executable specifications. Sometimes the "program behavior should be constrained but not uniquely determined" [Guttag78b]. Thus specifications alone may not be sufficient for the oracle function. Construction of an oracle to determine OK(d) can be very expensive [White87]. In testing research the existence of a test oracle is often assumed in order to refer to test cases to being correct or incorrect. One of the significant advances of our work is the fact that our testing method does not have to assume the existence of a test oracle.

A test set T can be viewed as a subset of the domain D of the software to be tested. The elements t of T ($T \subseteq D$) can be called test cases. Goodenough and Gerhart [Goodenough77] introduce the idea that test cases are chosen to satisfy some data selection criterion Dsc, where Dsc is a predicate over subsets of D. T is a test set if and only if Dsc(T). They define two desirable properties for a criterion Dsc: *reliability and validity*.

Definition (Reliable):

$$Reliable\,(Dsc) = (\forall\, T_1, T_2 \subseteq D)\,[\,(Dsc\,(T_1) \wedge Dsc\,(T_2)) \rightarrow (Successful\,(T_1) \leftrightarrow Successful\,(T_2))\,]$$

where $Successful\,(T) \equiv \forall t \in T \;\; OK\,(t)$.

Definition (Valid):

$$Valid(Dsc) = (\forall d \in D)[\neg OK(d) \rightarrow (\exists T \subseteq D)(Dsc(T) \rightarrow \neg Successful(T))].$$

A criterion is reliable if all test sets that satisfy that criterion produce the same result. A criterion is valid if at least one of the test sets that satisfy it produces the correct result. A criterion that is reliable and at the same time valid, is said to be an *ideal* criterion.

Howden [Howden78] showed that it is not possible to algorithmically produce a criterion that is guaranteed to be ideal. For all programs, passing an ideal test that satisfies an ideal criterion demonstrates correctness. Bouge [Bouge85a] shows that "extrapolating testing to infinity" (infinite cost, infinite knowledge) "is to show correctness with certainty." Thus testing may be viewed as a special case of proving correctness with certain restrictions on the kind of proof that is used.

2.1.1. Specification Directed Testing

In order to more formally describe this testing process we will use the mathematical tools as presented in [Gourlay83] and [Bouge85a] and consider first order languages.[3]

A language[4] L can be identified with its set of symbols. If S is a set, $L(S)$ is a

---

[3] "The first order language given by an alphabet consists of the set of ALL formulas constructed from the symbols of the alphabet." [Lloyd84]

[4] Language: "Any set of strings over an alphabet Σ, that is, any subset of Σ^* is called a language." [Lewis81]

language obtained by adding to L the members of S as constants. An extension of a language is denoted by $\subseteq$. If $L \subseteq L'$, then $L(S) \subseteq L'(S)$. A theory T on a language L is a set of $L(S)$-formulas (axioms). T is a finite theory if it contains a finite number of axioms. For example, if we have an alphabet Σ= {+, -, x, +} then our language L could be Σ^*. If we let S be the set of integers I, then our $L(S)$ language is $\{+,-,x,+\}^* \cup I$. We could *extend* L to L' to include "valid" where valid(A + 0)=false. Now $L \subseteq L'$ and $L(S) \subseteq L'(S)$. An *axiom* for our language L might be:

$$2\ 4\ +\ =\ 6.$$

A finite *theory* on L might be:

$$T=\{\ (2\ 4\ +\ =\ 6),\ \ (5\ 7\ .1\ +\ x\ =40)\}.$$

We are interested in the validation of implementations of abstract data types. ADTs are meant to be considered by users as "black boxes." Users are only concerned with the input/output behavior of ADTs, and not with their internal operations. Therefore ADT implementations may be modeled by their functional behavior. An ADT implementation may be described as a structure[5] [Bouge85a]. Our model of the implementation will be an L(S)-structure called "Π." Once we have represented our particular testing problem as an abstract testing context, we are dealing with a whole family of potential implementations (Π), to which the implementation under test belongs.

In Figure 1.1 we presented the testing process diagram as first presented by Bouge [Bouge85a]. The problem to be solved is: "Is the implementation correct with respect to its specification?" (Step Z + "run").This in turn breaks down to the three sub-problems of representation, construction, and application. In the representation phase (Step A of Figure 1.1) we build an abstract problem equivalent to the concrete one. This defines a testing context C.

Thus a *testing context* is defined as follows:

Definition (Testing Context):

> A testing context C is a 4-uple $\langle L, S, (\Pi), A \rangle$ where L is a first order language; S is a set; (Π) is a family of L(S)-structures; and A is an $L'(S)$-theory, where $L \subseteq L'$.

From this definition we see that the problem to be solved is stated as a testing context C that is made up of:

(1) a set of operations L,

(2) a set of values S on which those operations work,

(3) a family of implementations (Π) to which the one under test belongs,

(4) a set of properties A to be tested (the axioms of the algebraic specification).

Consider, as an example, a square-root program P, which should output the square-root of any natural number given as input. Then a reasonable testing context

[1] L(S)-structure: In this thesis an L(S)-structure may be viewed as an ADT implementation. (See glossary for definition)

C=⟨L,S,(Π),A⟩ could be:

L = {$ROOT$; $\cdot^2$; $\cdot \leq \cdot$ < ; $\cdot + \cdot$}, where the $ROOT$ symbol represents the function of the implementation
S = { The natural numbers }
(Π)= a family of implementations that provide the function "$ROOT$"
A = {$\forall x[ROOT(x)^2 \leq x < (ROOT(x)+1)^2]$}

Note that for a testing context ⟨L,S,(Π),A⟩ every structure in the family (Π) will also validate $L'(S)$-theories other than A. If we call such a theory H, we say C is an H-context if every element of (Π) validates H.

A testing context is an abstract model of a concrete testing problem. In this model the problem of testing an ADT is reduced to seeking an acceptable collection of test sets for a testing context [Bouge85b]. To define such a test set previous researchers [Bouge85b, Choquet86, Wild86, Bouge86] have made two "powerful" assumptions called a regularity hypothesis and a uniformity hypothesis.

For the *regularity hypothesis*, let us assume it is possible to associate a level of complexity with every series of function calls. A common complexity metric is the *number* of function calls in the series. The regularity hypothesis states that the axiom to be tested behaves "regularly" [Bouge86] with respect to the complexity measure. This means that as tests get more and more complex there is a point beyond which there is no point in testing because those more complex tests do not increase our confidence in the correctness of the implementation. For practical purposes, this

means that they assume there exists an upper bound on complexity beyond which testing is not needed. If one tests all combinations of function calls whose complexity is less than that upper bound and the system being tested behaves according to its specification for all those tests, then the implementation will behave correctly for any combination of function calls. For example, if a system works correctly for all lists shorter than some upper bound then it works correctly for lists of all lengths.

A complexity metric can be any function that can be applied to all possible test cases and then returns a numeric result. A complexity metric should give the user some measure of how "complex" a series of function calls is to test. The term "complex" can refer to syntactic complexity, computational complexity, space complexity, or any other sensible measure. Typically, complexity is measured by the number of function calls in the test case [Bouge85b, Choquet86].

For white box testing, Weyuker and Ostrand [Weyuker80] showed the need to partition the domain of a software system into sub-domains that are uniform with respect to correctness. A subdomain is uniform with respect to correctness if the correctness of one element of the sub-domain implies the correctness of all the elements of the sub-domain. Therefore, a uniform sub-domain D_i of an input domain D is such that $[(\exists d \in D_i \ OK(d)) \rightarrow (\forall d \in D_i \ OK(d))]$. For black box testing it has been shown that:

"If no complexity measure is available, we are faced with the well-known problem

> of partitioning variable domains in such a way that the axiom under test behaves uniformly on these sub-domains." [Bouge85b]

Bouge showed that this uniformity hypothesis is satisfied for each sub-domain when *there exists an element in each sub-domain such that a test of that element can infer the correctness of the whole sub-domain*. This is a powerful hypothesis and has not been sufficiently justified in previous works [Bouge85b, Choquet86, Wild86, Bouge86]. New regularity and uniformity hypotheses will be formally presented and justified in Sections 4.5 and 6.1 respectively.

A test for a given problem (given a testing context) is a finite set of experiments. An experiment is a question about the implementation whose answer can be answered in finite time. An example of an experiment for the ADT specified in Figure 2.1 might be:

"Does Frontq(Addq(Newq,4)) Return the value 4?"

As Bouge [Bouge85a] points out, this distinction between experiment and test is simply a matter of convenience as a test can always be identified with the conjunction of the experiments it contains.

A battery of tests should be a family of tests, ordered by some criterion. We can view a battery of tests as a sequence of tests: $T_1, T_2, T_3, \ldots, T_n, T_{n+1}, T_{n+2}, \ldots$ Thus we can define a *battery of tests* as follows:

Definition (Battery of Tests):

> Let C = $\langle L, S, (\Pi), A \rangle$ be a testing context. A battery of tests T for C is a pair $\langle H, (T_n)_{n \in N} \rangle$ where N is the natural numbers and H is an L(S)-theory such that C is an H-context; $(T_n)_{n \in N}$ is a family of tests for C such that for all $n \in N$ $T_{n+1} \cup H \rightarrow T_n$.

Recall that H is a theory about the testing context. The most commonly used H-theory is "T_n is a subset of the experiments in T_{n+1}." Now clearly *under this assumption*, if you pass all the experiments in test T_{n+1} you pass all the experiments in T_n, and $T_{n+1} \rightarrow T_n$. (T_n) has often been used instead of $\langle H, (T_n)_{n \in N} \rangle$ for the sake of conciseness. We will also use (T_n) when it is convenient to do so.

Goodenough and Gerharts's concepts of reliability, validity and ideality have been extended for specification-directed testing to *projective reliability, asymptotic validity* and *acceptability* by Gourlay [Gourlay83] and Bouge [Bouge85a].

Projective reliability refers to the consistency of results produced without regard to their usefulness. A family of tests is *projectively reliable* if passing a set of tests T_{n+1} implies passing the set of tests T_n.

Definition (Projective Reliability):

> Let C = $\langle L, S, (\Pi), A \rangle$ be an H-context, and $(T_n)_{n \in N}$ a countable family of tests for that context. $(T_n)_{n \in N}$ is projectively reliable if for every $n \in N$ and for every structure Π of (Π) such that $\Pi \rightarrow T_{n+1}$ then $\Pi \rightarrow T_n$.

We note that a battery of tests for a given context is projectively reliable because by definition $T_{n+1} \cup H \rightarrow T_n$ for a battery of tests $<H, (T_n)_{n \in N}>$. As we have noted, the most common H-theory is that test T_{n+1} contains all the experiments in tests $T_1, T_2, \ldots, T_n$. Given the knowledge (H) we know that passing all the experiments in T_{n+1} implies passing all the experiments in T_n.

Goodenough and Gerhart said that a test criterion is *valid* if, whenever the program is incorrect, the program will fail at least some test. This concept has been extended to asymptotic validity.

Definition (Asymptotic validity):

> Let $C = \langle L, S, (\Pi), A \rangle$ be a testing context, and $T = \langle H, (T_n)_{n \in N} \rangle$ a battery of tests for that context. T is asymptotically valid if for every structure Π of (Π) if $\Pi \rightarrow T_n$ for every $n \in N$ then $\Pi \rightarrow A$.

Since a battery of tests is necessarily projectively reliable, an *acceptable* battery of tests can be defined as follows:

Definition (Acceptable):

> Let C be a testing context, and T a battery of tests for that context. T is said to be acceptable if it is asymptotically valid and unbiased[6].

[6] Unbiased: A test is said to be unbiased if whenever the implementation to be tested is correct it will pass that test.

2.1.2. Summary of the Testing Process

In the representation phase of the testing process diagram in Figure 1.1 we produce an abstract testing context. We then construct an acceptable battery of tests for that context. Having obtained an abstract battery of tests we can pick some test T from it, according to a quality/cost assessment, and apply it to the given problem, leading to the conclusion.

By using the notions of a testing context and a battery of tests, these new definitions of projective reliability, asymptotic validity, and acceptability extend Goodenough and Gerhart's notions of valid, reliable and ideal to allow an asymptotic approach to the notion of testing. Based on the definitions presented here, we view a good test run as the application of a test chosen from an acceptable battery of tests for a testing context representing that particular testing problem.

2.2. Abstract Data Types

A data type is defined to be a pair <V,O> where V is a possibly infinite set of values, and O is a set of operations defined on those values. Thus a data abstraction requires the definition of a carrier[7] V, which is a set of values, and the definition of a set of operations defined on those values. An abstract data type defines a class of abstract objects that is *completely characterized by the*

[7] Carrier: The carrier of an algebra "is the set of mathematical objects we wish to manipulate, such as integers, real numbers or a set of character strings." [Stanat77]

operations available on those objects [Liskov74]. Data abstraction is useful because it allows us to define and use the essential concepts of a data type while hiding information about its implementation. Some examples of systems that might be implemented as abstract data types are: complex numbers, queues, or sorted lists. More complex items such as system activation records, and editors [McMullin83] have also been implemented as abstract data types.

An algebraic specification of an abstract data type will usually have two sections: a syntax section and a semantics section. The syntax section defines the data types of the input and output values of each of the functions of the abstract data type being specified. The semantics section is normally a list of axioms which together specify all the properties of the functions of the type being specified. One of the data types in the specification of an ADT is specified solely in terms of other types, that type is called the *Type Of Interest* and is denoted as the *TOI*. Figure 2.1 is an example of an algebraic specification of a queue. Appendices I and III contain larger examples of algebraic specifications of abstract data types.

For abstract data types, the value of a data object of the TOI is determined by the sequence of function applications which generate it from some initial state. This sequence is called a "trace" . The value of a data object is also

Type Queue(Integer)

SYNTAX

Newq	-> Queue
Addq(Queue,Integer)	-> Queue
Deleteq(Queue)	-> Queue
Frontq(Queue)	-> Integer U {error}
Isnewq(Queue)	-> Boolean

SEMANTICS

For all q : Queue; i : Integer , Let

Isnewq(Newq)	- True
Isnewq(Addq(q,i))	- False
Deleteq(Newq)	- Newq
Deleteq(Addq(q,i))	- IF Isnewq(q) THEN Newq ELSE Addq(Deleteq(q),i)
Frontq(Newq)	- error
Frontq(Addq(q,i))	- IF Isnewq(q) THEN i ELSE Frontq(q)

END Queue

Figure 2.1 An Algebraic Specification of a Queue

called an "instance" of the data type. For example

Addq(Addq(Addq(Newq,4), 10), 6)

produces an element of the TOI (a queue) with three elements and those elements are the integers 4, 10, and 6. All instances of a TOI whose traces differ only in the values of the non-TOI input variables are said to form an instance class. In this case replacing the integers 4, 10, or 6 with other integers would produce another queue in the same instance class. Thus an instance class can be viewed as a generalization of an instance. For example: the trace

Addq(Addq(Addq(Newq,Variable1), Variable2), Variable3)

could produce two different queues containing the values [123, 66, 99], and [747, 1, 999]. Both of these queues would belong to the same instance class.

Definition (Instance):

> Let L be language and S a set. An instance i of L(S) is an L(S)-formula without quantifier or logical symbols whose symbols are $p_1, p_2, p_3, \ldots, p_n$ where n is the number of L(S) symbols in i.

Definition (Instance Class):

> Let L be a language and S a set. An instance class I of L(S) is formed from an instance $i = p_1, p_2, p_3, \ldots, p_n$ of L(S) by including all instances $i' = p'_1, p'_2, p'_3, \ldots, p'_n$ of L(S) that have the same number of symbols as i and where if p_x is different from p'_x then $p_x \in S$ and $p'_x \in S$.

We can classify the types of functions in an algebraic specification of an abstract data type by the type of values they accept and return. Functions that

return values that are not of the TOI are called O-Type (Output) functions. For our queue example "Frontq" is an O-type function. Those functions that return a value of the TOI but do not accept any arguments of the TOI are called N-Type (New) functions. "Newq" from our example is an N-Type function. Those functions that accept a value of the TOI and return a value of the TOI are either C-Type (Constructor) or E-Type (Extendor) functions. The difference between C-Type and E-type functions is:

> any trace containing an E-Type function application can be rewritten in an equivalent form containing only N-Type and C-Type function applications.

For the queue example given in Figure 2.1 "Deleteq" is an E-Type function because any trace containing a "Deleteq" can be equivalently written as a series of "Addq's" and "Newq's". For example

Deleteq(Addq(Addq(Addq(Newq,1), 2), 3))

is specified to be equivalent to

Addq(Addq(Newq,2), 3).

It is important to note that since the TOI for an abstract data type is defined solely in terms of other types and the operations that may be applied to it, the functionality for that abstract type is totally defined by the output of the O-type functions.

Definition (Functionality of an Instance)

> Let i be an instance of a language L(S), and a L(S)-structure Π be an implementation. The functionality of i in Π is the set of pairs $\langle o_* \in O \ , v_* \rangle$, where O is the set of O-type operations in L that may be applied to i such that their meaning in Π is calculable, and v_* is that meaning in Π.

Definition (Observable Functionality of an Implementation):

> Let L(S) be a language, let an L(S)-structure Π be an implementation, and let T be a set of instances of L(S). The observable functionality of Π for T is the union of the functionalities of all instances i ∈ T in Π.

If we build an abstract data type with no O-type functions then that ADT will be viewed as having only one value, as the user will have no way of differentiating two different values of the TOI. Thus any useful abstract data type must have at least one operation that returns a value that is not of the type of interest. This leads to an answer to the question "what is *abstract* about an abstract data type?" An abstract data type is *abstract* in that from the user's point of view it is an isomorphism class of types rather than any particular *concrete* representation of the class. From a tester's point of view we are working with a whole family of potential implementations (Π) to which Π ,the implementation under test, belongs.

Abstract data types facilitate software construction by successive decomposition. At any point in the decomposition we can view the programming task as writing a program which:

(1) solves the problem;

(2) runs on an "abstract machine" [Liskov74] which provides those data objects and operations that are ideally suited to solving the problem.

Note that abstract data types can form hierarchies where "higher" types are built on top of and using "lower" types. In such a case the lower types are part of the abstract machine upon which higher types are executed. This process of building ADTs on top of other ADTs which are in turn built on top of other ADTs may be continued for many levels. At each level the lower types are part of the abstract machine upon which the new ADT is built.

While our queue example is a simple one, it should not be concluded that only simple software systems can be viewed as abstract data types. A messaging system [Roberts88] and a line editor [McMullin82] are examples of software systems that have been specified as a set of permitted operations and a set of abstract objects that are defined in terms of those permitted operations. We have also seen an initial attempt at defining the file system of the UNIX operating system as an abstract data type [Gaudel88].

There is a final point about abstract data types that is often forgotten or left unstated, but which we believe is significant to our work and any other work related to testing them. Guttag [Guttag80] has found that, with some notable exceptions, types with four or less C-type and N-type functions are manageable, but

that types with more than four constructors are usually more easily specified by decomposing them into simpler types. Therefore, we believe the examples we present in this thesis, while not huge, are sufficient to show the applicability of our method.

2.2.1. Implementation of Abstract Data Types

The most obvious method of implementing an ADT, given an algebraic specification, is to use the axioms in the specification itself as function definitions, to obtain an "automatic" implementation. Guttag [Guttag78b] and Moitra [Moitra79] have shown that given a restricted set of axioms, it is possible to produce a crude implementation from algebraic specifications. Berztiss [Berztiss83] shows that this is not always possible and not often desirable. The basic problem is that the kinds of axioms allowed in algebraic specifications are too general to be used as simple re-writing rules. "For example, the arbitrary level of nesting on both the left- and right-hand sides of the axioms necessitates complex pattern matching to determine applicability. Commutativity axioms can be used for re-writing only under heuristic guidance for fear of looping indefinitely" [Berztiss83].

In practice the implementation phase of ADT development involves two steps: the choice of a representation of the abstract objects and the implementation of the abstract operations in terms of that representation. The design process begins with a formal, declarative specification of the problem. An algebraic specification is an

example of such a specification. This is followed by a series of procedural and data refinement stages that yield a series of programs, the last program in the series is the final code [Laski88].

Essential to this refinement process is the notion of the *abstraction* function that maps a concrete object into an abstract one. Such a function defines an interpretation, i.e., the abstract meaning of the data [Laski88]. For example, assume we implement the queue specified in Figure 3.10 with an array of integers (to store the values) and two natural numbers (to store where the top and bottom currently are). In this case there exists an abstraction function that converts a concrete array of integers and two natural numbers into an abstract entity we call a *queue*. In general, when implementing a type T, using objects $t_1, t_2, \ldots, t_n$ one needs an abstraction function, say *Abstr*, with the functionality

$$Abstr: \ t_1 \times t_2 \times \cdots \times t_n \rightarrow T.$$

This abstraction function does not need to be specifically written down. Indeed, it often only exists in the programmer's mind.

The existence of an abstraction function has some significance for software testing. As we have shown, implementations and specifications of ADTs can be treated as algebras. The testing problem can now be viewed at an abstract level, as one of showing that this abstraction function which is a mapping from one algebra

to another, is a homomorphic[8] mapping from one to the other.

2.3. Test Case Generation Using PROLOG

Several authors [Wild86, Bouge85b, Pesch85] have reported the methodology of translating algebraic specifications of abstract data types into PROLOG horn clauses and then using them to generate test cases. In this section we will review the use of the algebraic specifications as a guide to allow a PROLOG interpreter to produce test cases. It should be emphasized that the methodologies outlined in this section are currently only being directed toward testing of abstract data types.

2.3.1. The Test Procedure

While the various research groups looking at using PROLOG to test implementations of abstract data types [Wild86, Bouge86, Pesch86] vary in the degree of formality with which they approach the problem, the basic testing procedures they follow are very similar. Their general approach can be outlined as follows:

(1) Choose an instance class from the set of instance classes of the TOI (recall this is a trace of function applications).

(2) Instantiate all input variables to constants, to form a particular value "V."

[8]Homomorphism: A structure preserving mapping between algebras. (See Glossary)

(3) Apply the sequence of function applications specified in "V" to the implementation to produce the value "Vi."

(4) Apply the sequence of function applications specified in "V" to the specifications to produce the value "Vs."

(5) All O-type functions are applied to Vi. This results in an implementation output being returned from Vi.

(6) If the implementation output obtained in step (5) is the same as the specified output obtained from Vs, then the test succeeds; otherwise it fails.

2.3.2. Using PROLOG

It should be noted that the specification of the abstract data type must be in such a form that we can apply function specifications to obtain the specified result value "Vs." This is where PROLOG enters into the process. It has been noted by various authors [Wild86, Bouge86, Bouge85b, Choquet86, Pesch85] that the axioms (function definitions) of an algebraic specification of an abstract data type can be mechanically translated to the horn clauses that make up a PROLOG program. As [Choquet86] points out, an axiom of the form :

$$a(x)=b(x)=>t(x)=t'(x)$$

can be written under a PROLOG-like formalism as :

$$t(x)=t'(x):-a(x)=b(x).$$

The transformation of an algebraic specification into a horn clause is performed by using the following equivalences due to Bouge [Bouge86]:

1) $\cdots g(x_1,\ldots,x_n)=y \cdots \quad \Leftrightarrow \quad \cdots G(x_1,\ldots,x_n,y) \cdots$
where G is a relation symbol corresponding to the n-ary defined operator g and $x_1,\ldots,x_n,y$ are variables.

2) $P(g(x_1,\ldots,x_n),z_2,\ldots,z_m) :- \cdots \quad \Leftrightarrow \quad P(y,z_2,\ldots,z_m) :- g(x_1,\ldots,x_n)=y \cdots$
for any m-ary relation symbol P, any n-ary defined operator g, any argument position of $g(x_1,\ldots,x_n)$.

3) $\cdots :- P(g(x_1,\ldots,x_n),z_2,\ldots,z_m) \cdots \quad \Leftrightarrow \quad \cdots :- P(y,z_2,\ldots,z_m), g(x_1,\ldots,x_n)=y \cdots$
for any m-ary relation symbol P, any n-ary defined operator g, any argument position of $g(x_1,\ldots,x_n)$.

To see how these equivalences work, note how the first equivalence would translate the axiom

Isnewq(Newq) = True

into the following horn clause:

isnewq(newq,true).

Figure 2.2 gives a listing of a PROLOG translation of the queue specification given in Fig. 2.1.

```
isnewq(newq, true) :- !.
isnewq(addq(Queue,I), false).

frontq(newq, error) :- !.
frontq(addq(newq, I), I) :- !.
frontq(addq(Queue, I), X) :- frontq(Queue, X).

deleteq(newq, newq) :- !.
deleteq(addq(newq,I), newq) :- !.
deleteq(addq(Queue, I), addq(X, I)) :- deleteq(Queue, X).
```

Figure 2.2 PROLOG Specification of a Queue.

Once the specifications have been written in PROLOG, we can use a PROLOG interpreter to do several things.

I PROLOG will automatically perform step (4) of the testing process listed in Section 2.3.1. That is, given a set of specifications and an instance class, a PROLOG interpreter will return the general specified result "Vs." It should be noted that the PROLOG interpreter is acting as an oracle for a class of test cases.

II PROLOG will automatically perform the instantiations required by step (2) of the testing process. In so doing it will partition the given instance class

into sub-instance classes if any such sub-instance classes exist. We will discuss this partitioning in Section 2.3.4. The differentiation of these sub-instance classes depends on the properties of the input variables. For example, if the behavior of the TOI depends on whether I1 = I2 or not, a PROLOG interpreter would give results (act as an oracle) for both the I1 ≠ I2 and I1 = I2 cases and give the constraints on I1 and I2 for the I1 = I2 case. In this case that would mean stating that I1 and I2 must be the same.

III If a given set of function applications is infeasible (possibly due to an impossible requirement on the input values such as I1=I2 AND I1 ≠ I2), then the PROLOG interpreter will simply return "no" if asked to instantiate the necessary values for a test case.

2.3.3. Sample Run

How to actually generate test cases is best described by an example. Recall that all we are doing is implementing the final few steps of the test procedure outlined in Sub-section 2.3.1.

The example given in Figure 2.3 is taken from [Wild86] and shows the generation of the test cases and correct results according to the specifications for a few simple instance classes for the queue example.

```
2) ?- deleteq(newq,Answer).
Answer = newq
3) ?- frontq(newq,Answer).
Answer = error
4) ?- isnewq(newq,Answer).
Answer = true
5) ?- deleteq(addq(newq,I1),Answer).
I1 = _0
Answer = newq
6) ?- frontq(addq(newq,I1),Answer).
I1 = _1
Answer = _1
7) ?- isnewq(addq(newq,I1),Answer).
I1 = _2
Answer = false
```

Figure 2.3 Generating Test Cases and Results .

EXPLANATION:

(1) Queries 3 and 4 define all the test cases (frontq and isnewq are the only O-type functions) that should be applied to an empty queue. Note also that the answer the implementation should return is also given ("error" and "true").

(2) For query 6 the PROLOG interpreter implies that to add an integer to a queue and then apply "frontq" to that queue, we supply a value for

that integer and the result from applying "frontq" should be that value.

(3) Query 7 implies that for the O-type function "isnewq" there must be a value for that integer but the result should be "false" no matter what value that integer is given.

Just as queries 3 and 4 make up all the test cases for an empty queue, queries 6 and 7 make up all the test cases for a queue of length one. We point out that while the example given here is simple, this methodology has been successfully applied to more complicated problems [Choquet86].

2.3.4. Partitioning of Instance Classes

In Section 2.3.2 we stated that "PROLOG will automatically perform the instantiations required by step (2) of the testing process. In so doing, it will *partition* the given instance class into sub-instance classes if any sub-instance classes exist." A sub-instance class would exist if the instance class given was not specified to be functionally uniform across the domain of all the non-TOI input variables.

Definition (Input Values):

> Let L be a language, S be a set, and I be an instance class of L(S). For an instance $i = p_1 p_2 p_3 \ldots p_n \in I$ we obtain the sequence of input values $q_1 q_2 q_3 \ldots q_m \quad (m < n)$ for i by removing all symbols p_x in i where $p_x \notin S$.

Definition (Sub-Instance Class):

> Let L be a language, S be a set, I be an instance class of L(S), and A be an L(S)-theory. For an instance i= $p_1,p_2,p_3,\ldots,p_n \in$ I with input values $q_1,q_2,q_3,\ldots,q_m$ there exists a function F_i without conditionals such that according to A $F_i(q_1,q_2,q_3,\ldots,q_m) \equiv$ i. A sub-instance class I_i of I is formed from i by including all instances $i_x \in$ I with input values $q_{x1},q_{x2},q_{x3},\ldots,q_{xm}$ such that according to A $F_i(q_{x1},q_{x2},q_{x3},\ldots,q_{xm}) \equiv i_x$.

Such sub-instance classes arise in an ADT when *"what the operations do"* is dependent on the values of some of the non-TOI variables. In our queue of integers (Fig 2.1, Fig 2.2) such sub-instance classes do not exist, as none of the axioms are dependent on the values in the queue.

To illustrate sub-instance classes and how a PROLOG interpreter will handle them, we extend our queue example. We introduce the function:

has (Queue, Integer): BOOLEAN

This function will return true if the given *Queue* has the value *Integer* in it, and false otherwise. The additional axioms would be:

```
Has (Newq,j)          -False
Has (Addq(q,i),j)     -IF i=j THEN True
                      ELSE Has(q,j)
```

This would create the following three additional PROLOG horn clauses:

```
has(newq,J,false) :- !.
has(addq(Queue,I),J,true) :- I=J.
```

```
has(addq(Queue,I),J,Answer) :- has(Queue,J,Answer).
```

Now we will have an additional query for queues of length one. This query will arise from applying the O-type function *has* to that queue. Figure 2.4 extends our example from Section 2.3.3 with a new query 8).

```
6) ?- frontq(addq(newq,I1),Answer).
I1 = _1
Answer = _1
7) ?- isnewq(addq(newq,I1),Answer).
I1 = _2
Answer = false
8) ?- has(addq(newq,I1),I2,Answer).
I1 = _3
I2 = _3
Answer = true ;

I1 = _3
I2 = _8
Answer = false
```

Figure 2.4 Extended Generation Example

It is important to note that for query 8), PROLOG returned more than one result (in this case two). These two results are produced because in this case, there are two sub-instance clauses for that particular instance class:

input values: q_1 = I1; q_2 = I2
$F_a(q_1,q_2)$ = true
$F_b(q_1,q_2)$ = false.

Thus depending on the values of the non-TOI input variables (in this case I1 and I2), we get two separate output functions. Both of these cases must be tested. The important point for our work is that if PROLOG is given algebraic specifications in an *appropriate* form and a trace in an *appropriate* form it can determine all the different possible output values for that trace. An *appropriate* form is determined by the equivalences given in Section 2.3.2.

2.3.5. Problems

This methodology is a useful tool as it provides an oracle and allows some automatic test case generation given an instance class. We foresee, however, several problems that may hinder this as a useful test case generation method:

(1) PROLOG's backtracking is depth-first and thus leads to potentially infinite depth-first recursion of instance classes.

(2) The PROLOG interpreters currently available can only distinguish behavior based on the equality of some input variables. Behavior based on inequalities is a problem. For example: given a domain D = { (x,y) | x ≠ y }, there should be only one instance class with a pair of variables and two sub-instance classes, one with a constraint stating that the variables are not equal,

and one with a constraint stating the variables are equal. Unfortunately a PROLOG interpreter can only enumerate all possible x's and all possible y's and then exhaustively list all the pairs (x,y) satisfying $x \neq y$.

(3) This testing procedure assumes the ability to generate all instance classes and to be able to choose which of the untested ones should be tested next. We do not have the ability to automatically generate all instance classes as there may be infinitely many of them. For the instance classes we can generate, there is no useful theoretical basis for determining which instance class should be tested next. Current methods [Choquet86, Bouge86, Wild86] simply select the shortest untested instance class available and test it.

3. T-3: A PROTOTYPE

In this chapter we describe our initial new software testing methodology. This methodology is based on the software specification. We also describe a prototypical system called T-3 (Type Testing Tool) that implements it. This initial new methodology is a combination and extension of the results of two separate research directions: test harness construction and using PROLOG to generate individual test cases. Background on both of these subjects is given in Chapter 2.

Previous results in test harness construction [Gannon81, McMullin83, Day85] and in the use of PROLOG to generate test cases [Bouge85b, Choquet86, Wild86, Bouge86] have a significant property in common. They both work on abstract data types. Thus our first decision for our specification-directed software testing system was to take advantage of the preliminary work done by other researchers and limit ourselves to abstract data types. As we outlined in Chapter 2, most software systems *can* be viewed as abstract data types, so we do not feel this is a significant limitation to the applicability of our work.

Since our methodology and prototype extend previous research, the following Sub-section gives a new analysis in terms of the testing process (Figure 1.1) for the DAISTS test system. In Section 3.2 we outline our new methodology and briefly describe the operation of T-3 in Section 3.3. A detailed outline of T-3 can be found in Appendix II.

3.1. Test Harnesses

For our automatable testing method and for our prototype implementation, it was necessary to develop a methodology for automatically producing test harnesses suitable to our needs. In terms of the testing process diagram (Figure 1.1), we need a method for performing step C (application).

Gannon et al. [Gannon81, McMullin83, Day85] have developed a compiler-based software system that combines a data-abstraction implementation language and ADT specification by algebraic axioms. This system, called DAISTS, is intended to help with the production of correct implementations of ADTs. It does this by providing a test harness for an abstract data type given the implementation, the specifications of the ADT, and a set of test data.

3.1.1. What Does DAISTS Do?

Given the axioms and the implementing code for an ADT, the DAISTS system produces a "program" that consists of the axioms as test drivers for the implementation. Test data in the form of expressions using the abstract functions and constant values are fed to this program to determine if the implementation is correct by determining if the axioms and implementation agree. Thus we have in essence an automated testing system; the user produces axioms, the implementation, and test data; DAISTS then writes the test drivers. Section 3.1.3

gives an example of how DAISTS and similar systems work.

3.1.2. How Does DAISTS Work?

There are three main parts to the input to the DAISTS system: an implementation of an abstract data type written in SIMPL-D, a collection of algebraic axioms describing the type, and a collection of test data. DAISTS produces a "program" containing :

(1) the implementing code compiled as if it had been produced by a debugging compiler;

(2) a block of code calling on this implementation; this code has been compiled from the supplied axioms of the specification;

(3) a driver program that cycles through the given data;

(4) a set of execution monitoring routines.

As this program executes, the test data drive the axioms, the axioms in turn drive the implementation code. After the execution a summary reports any data for which the axioms did not hold, as well as some coverage measures. Figure 3.1 gives a schematic description of the operation of the DAISTS system.

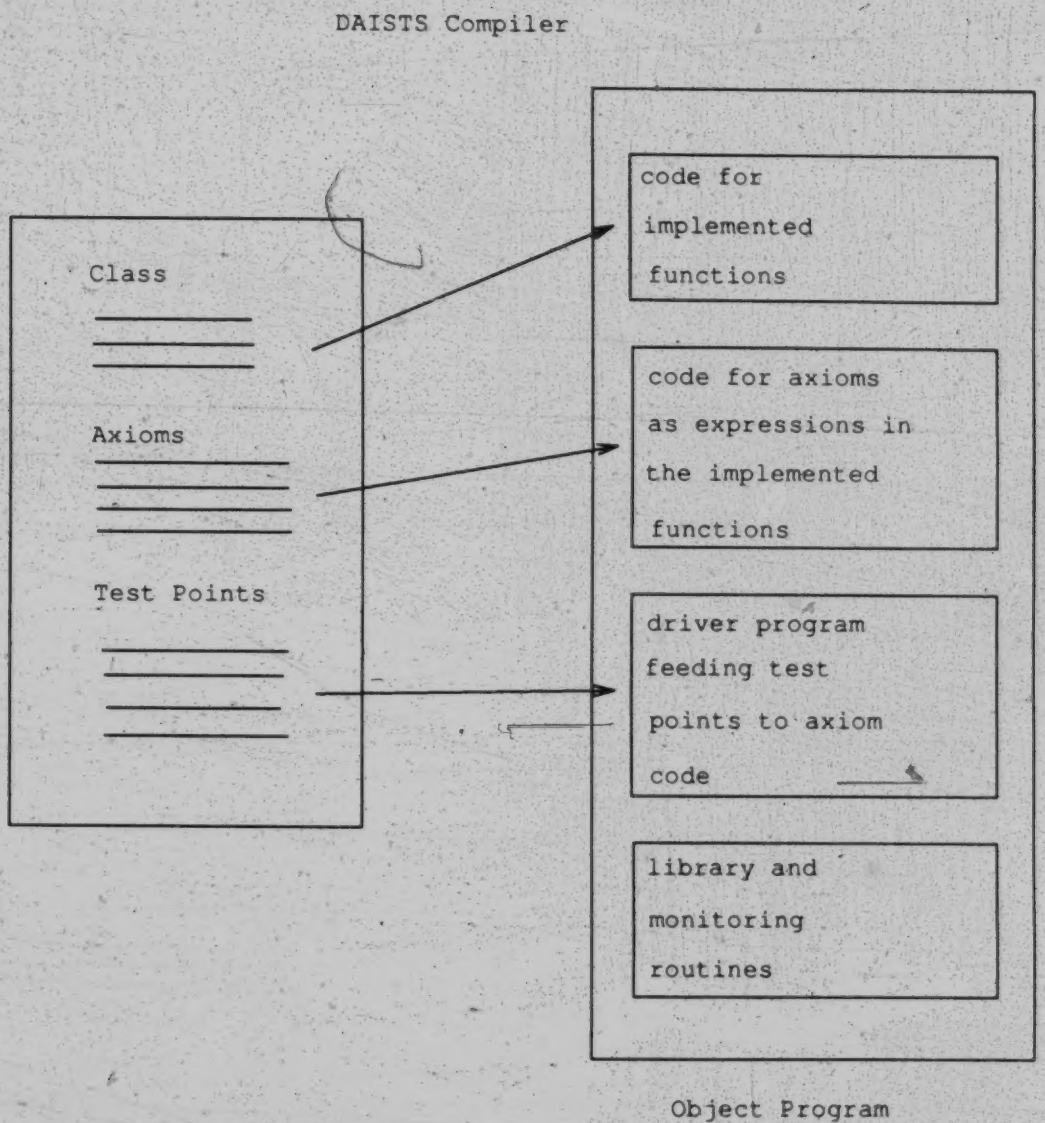


Figure 3.1 The DAISTS System.

There are four basic sections of the input to the DAISTS system. They are the "class definition" which implements the ADT, the "axioms," "testpoints," and "testsets." The general form of the input to the DAISTS system is given in Figure 3.2.

```
        .
        .       SIMPL-D class definition
        .
axioms
        .
        .       algebraic axioms describing the type
        .
testpoints
        .
        .       declaration and initialization of test values
        .
testsets
        .
        .       set of test points to be used with each axiom
        .
start
```

Figure 3.2. The DAISTS Input Stream

The implementation is written in a subset of SIMPL-D. The "testsets" section contains a list of axiom names with values to be substituted for the free

variables of the axioms. A common data object may be needed in testing several axioms in the "testsets" section. The "testpoints" section allows the user to build such objects just once so that they may be referenced in different portions of the "testsets" section.

3.1.3. An Example

The following example of the operation of the DAISTS system is due to McMullin et al. [McMullin82].

Given the input listed in Figure 3.3, DAISTS produces a procedure for the axiom "Pop1" as listed in Figure 3.4.

The DAISTS input listed in Figure 3.3 causes the code listed in Figure 3.5 to be produced within the main test driver program

```
                                        .
                                        .
                                        .
Axioms
Pop1(BoundedStack S, integer I):
        Pop(Push(S,I))=
                if Depth(S)= DepthLimit
                then Pop(S)
                else S

Testpoints
        BoundedStack S1, S2
        S1 := Push(Push(Push(NewStack,1),1),2)
        S2 := S1
        while Depth(S2)<DepthLimit
          do
                Push(S2,Top(S2)+Top(Pop(S2)))
          end

Testsets
Pop1:  (Newstack,7), (S1,22), (S2,83),
        (Push(Pop(S2),19),82)
                                        .
                                        .
                                        .
```

Figure 3.3: Sample DAISTS Input

```
proc Pop1(BoundedStack S, integer I, integer Testsetnumber)
        if StackEqual(Pop(Push(S,I)),
                exprif Depth(S) = DepthLimit
                        exprthen Pop(S)
                        exprelse S)
        then
                return
        endif
        call Report_axiom_failure('Pop1',Testsetnumber)
```

Figure 3.4 Procedure from an Axiom

```
                                .
                                .
                                .
call Pop1(NewStack,7,1)
call Pop1(S1,22,2)
call Pop1(S2,83,3)
call Pop1(Push(Pop(S2),19),82,4)
                                .
                                .
                                .
```

Figure 3.5 DAISTS Main Test Driver

Note how DAISTS compiles the algebraic axiom into a procedure that compares equivalent words by comparing the values returned by the implementation functions

that comprise the words. Each user-supplied set of test data is translated into a call to the procedure that was generated from the appropriate axiom.

3.1.4. Specifications

Since the DAISTS system attempts to mechanically determine the consistency of specifications and implementations a formal specification technique is essential [Gannon81]. The specification fed to the DAISTS system consists of a series of axioms. The primitives of the specification language include boolean and integer constants, free variables, equality, boolean and integer operators, and functional composition. T-3 uses a similar specification language because T-3 has the same need for a formal specification technique.

3.1.5. DAISTS within the Testing Process

When we compare the function of DAISTS with the testing process diagram in Figure 1.1, we see that DAISTS only performs the "application" function (Step C). In requiring the *a priori* existence of test points and test sets, DAISTS assumes that representation and construction have been done elsewhere. Therefore, to produce our automated test system (T-3) we have had to develop and incorporate methods for representation and construction. Figure 3.6 shows the operation of DAISTS within the testing process. Given a specification, implementation and a previously constructed set of test points and test sets, DAISTS will apply the test data to a SIMULA

program. That is, given the results of the representation and construction steps, DAISTS performs the application step of the testing process.

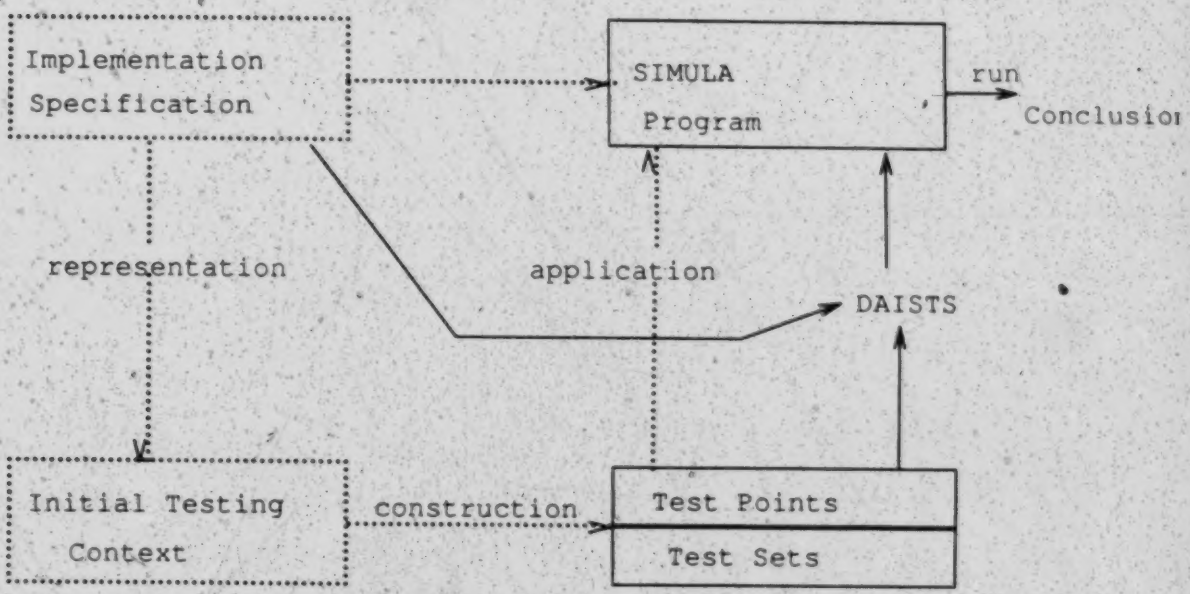


Figure 3.6: DAISTS within the Testing Process

3.1.6. Advantages

Normally, when a software system is tested, a result is obtained from the implementation and then checked against the "correct" result. This "correct" result

is supplied by a test oracle. Generally programmers serve as test oracles for software. They then compare the program output against their often imprecise and inadequate understanding of the problem being solved and agree too quickly with the results of the program execution [Day85]. One of the advantages of the DAISTS system is the fact that the correctness of the implementation is determined independent of any human decisions. This advantage is carried through to our testing system (T-3).

Gannon et al. [Gannon81] showed that using DAISTS helped users to approach testing in a more systematic manner. They found that more of the code was tested. They also found that the use of DAISTS did not lengthen the development process.

3.1.7. Constraints

It is often convenient and sometimes necessary to use hidden or auxiliary functions in the specification of an ADT. Such hidden functions cannot appear as part of programs using the ADT. They are part of the specification of the abstraction but not of the abstraction itself. The specification of a stack of at most 100 integers given in Figure 3.7 makes use of the hidden function "Depth." Depth is not a function the ADT supplies to the user. Depth is only used to make the specification simpler and easier to use. Now, even if we assume that the program is either correct or "close" to correct, and assume a top-down methodology was followed, then the

Type Stack(Integer)

SYNTAX

New	->Stack
Push(Stack,Integer)	->Stack
Pop(Stack)	->Stack
Top(Stack)	->Integer
Isnew(Stack)	->Boolean
Replace(Stack,Integer)	->Stack

SEMANTICS

For all s:Stack; i:Integer, Let

1) Pop(New)	- New
2) Pop(Push(s,i))	- IF Depth(s) ≥ 100 THEN Pop(s) ELSE s
3) Top(New)	- error
4) Top(Push(s,i))	- IF Depth(s) ≥ 100 THEN Top(s) ELSE i
5) Isnew(New)	- true
6) Isnew(Push(s,i))	- false
7) Replace(s,i)	- IF Isnew(s) THEN Push(s,i) ELSE Push(Pop(s),i)
8) Depth(New)	- 0
9) Depth(Push(s,i))	- 1+ Depth(s)

END Stack.

Figure 3.7: An Algebraic Specification with a Hidden Function: Depth

best we can say is that "most of the functions mentioned in the specifications will probably be implemented." We certainly cannot guarantee that all hidden functions in the specification will appear in the implementation. For example, for our type Stack in Figure 3.7 we cannot guarantee the function Depth will appear in the implementation. Thus DAISTS simply cannot test the ADT specified in Figure 3.7.

There are two main problems with the DAISTS system. The first problem is that the system requires each function MENTIONED (as opposed to just the ones being specified) in the specification to be implemented. That is, every hidden function in the specification must also exist in the implementation. This is of course by no means necessary for the implementation to be correct. The implementation is only REQUIRED to implement the functions that will be externally visible.

The second problem is that an equality function for each new type must be provided. In the example in Figure 3.4 the function "StackEqual" was required. This is a problem because the proper meaning of "equality" changes from one data type to the next. Thus a generic equality would seldom implement what the user intended. An equality function may not be required by either the specifications or the implementation of an ADT. Thus requiring an equality function leads to two problems for the DAISTS system:

(1) The DAISTS system requires the tester to understand the idea of equality for the new type, which may not necessarily be straight forward. The tester then has to implement this equality function.

(2) An error in this tester-defined equality function may cause the DAISTS system to state that there was an inconsistency between the specifications and the implementation when no such inconsistency existed. An even worse possibility is that such an error could lead the system to miss an inconsistency that did exist.

3.1.8. Experiences

McMullin, et al. [McMullin83] conducted a case study using the DAISTS system to specify, implement and test a record-oriented text editor. They found that producing formal specifications reveals interesting boundary conditions that are often omitted in even the most carefully constructed informal problem statements. Using the specifications with a testing tool forced their implementations to handle the boundary conditions .

While they found the use of algebraic axioms as a formal specification language unwieldy for several of the functions that they implemented, they considered this test of the DAISTS system successful. They found that only two errors escaped unit testing, and one of those was due to the specification and

implementation both being wrong.

3.2. A New Method for Testing ADTs

In the previous section we discussed how Gannon, et al. [Gannon81, McMullin83, Day85] have developed a compiler-based software system that can provide an automated testing of an abstract data type implementation given the specifications of the abstract data type, the implementation itself, *and a set of test cases*. Several authors [Wild86, Bouge85b, Pesch85] have reported a methodology of translating algebraic specifications of abstract data types into PROLOG horn clauses and then using them to generate test cases (See Section 2.3). Our initial methodology for testing ADTs builds on these two earlier methodologies and is an improvement on both, as it will automatically test an abstract data type implementation given only the implementation and the specification. The software tester need not produce a set of test cases as was the situation before. The following is the high level outline we initially used to develop our testing methodology.

(1) Follow the methods described by Bouge et al. [Choquet86, Bouge86, Wild86] to produce a series of instance classes to be tested . These methods use a PROLOG interpreter.

(2) Select a set of concrete instances from that instance class.

(3) Produce MODULA-2 code to run all the test cases.

(4) Run that code.

Figure 3.8 shows this high level operation of T-3 in terms of the testing process diagram. The abstract steps of representation and construction are executed by producing a series of instance classes in step (1). The concrete step of applying a test set is broken down into two parts: selecting the particular tests to run (step(2)) and producing the MODULA-2 code to test them (step(3)).

In Step (1) we use the equivalences we gave in Section 2.3 to produce a PROLOG version of the specifications. We then generate instance classes for the ADT starting with the shortest and then producing all successively longer instance classes. Bouge [Bouge85a] has shown that this ordering of instance classes ensures that the tests produced are *acceptable* (asymptotically valid, projectively reliable, and unbiased).

Step (2) involves sending the instance classes to a PROLOG interpreter whose program is the PROLOG version of the specifications. This procedure is described in Section 2.3 and elsewhere [Wild86, Choquet86, Bouge85b].

In Step (3) we apply all applicable O-type functions to the concrete instances and compile these with the algebraic axioms for those O-type functions to produce MODULA-2 code. Figure 3.9 shows the MODULA-2 code produced for the Queue type specified in Figures 2.1, 2.2, and 2.3 for the concrete test instance addq(newq,8).

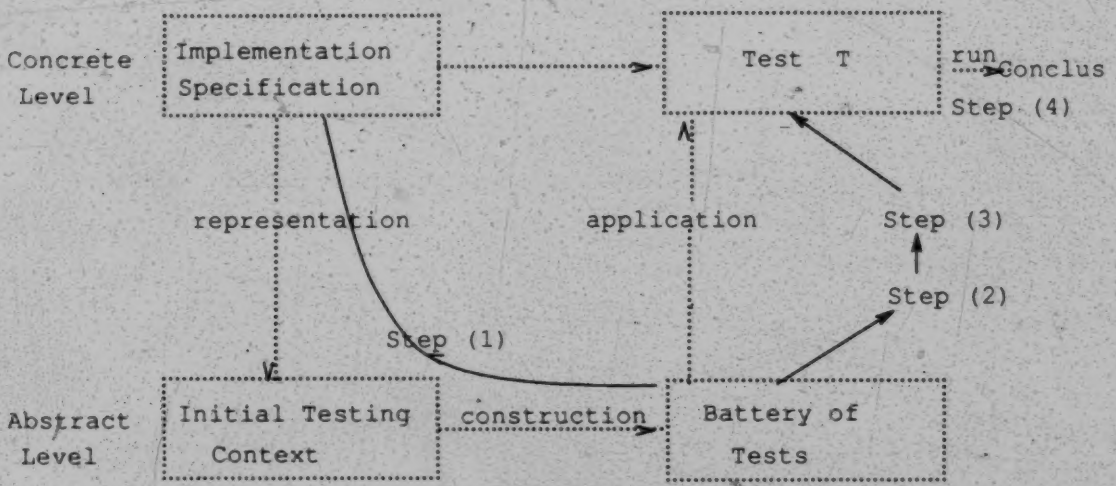


Figure 3.8: T-3 Testing Process

Note these two tests correspond to queries 6 and 7 in Figure 2.3.

3.2.1. Specification Format

The form of the algebraic specifications to be used in our methodology is similar to that outlined by Guttag [Guttag78b]. We have chosen this type of format because we have observed that recent publications on the use of abstract data types have, to varying degrees, adopted this as a standard format.

Instance:	addq(newq(),8)
O-type functions	isnewq,front

```
MODULE              MainTest

FROM Queue IMPORT
        queue,
        newq , add , remove , front , isnewq ;

FROM StdMonitoring  IMPORT
        Failure , Pass;

BEGIN
                                        .
                                        .
                                        .
ELSEIF (NOT(front(add(newq(),8)) = 8 ))
        THEN
                Failure(11)
ELSEIF (NOT(isnewq(add(newq(),8)) = FALSE ))
        THEN
                Failure(12)
                                        .
                                        .
                                        .
ELSE
        Pass()
END;

END MainTest.
```

Figure 3.9: MODULA-2 Code to Test an Instance

(1) For our specification language we will assume five primitives: functional composition, an equality relation (=), two distinct constants (TRUE & FALSE) and an unbounded supply of free variables. From these primitives it is possible to construct an arbitrarily complex specification language, because once we have defined an operation in terms of these primitives it may be added to the language. For example, an IF-THEN-ELSE operation may be defined as follows:

IF-THEN-ELSE (TRUE, q, r) = q
IF-THEN-ELSE (FALSE, q, r) = r

and thus IF-THEN-ELSE may be added to the language of the five primitives.

(2) We will assume that the operation IF-THEN-ELSE(A,q,r) shall be part of the specification language. This operation shall be of the form:

IF A THEN q ELSE r.

(3) We shall also assume the availability of the standard BOOLEAN operators: AND, OR, NOT.

(4) For simplicity we also allow for the type INTEGER and the conventional operations on integers: +, -, *, /, mod ≥, <, ≠.

We realize that a richer language may often be more desirable for the specification of an abstract data type. We have restricted our specification language for two basic reasons.

(1) Simplicity: With a limited number of specification language constructs we allow for simpler translation of algebraic axioms to PROLOG and a simpler analysis of the test cases that need to be run.

(2) Sufficiency: Our simple language is sufficient to specify any abstract data types we want. We observe that all conventional programming control constructs can be translated into our basic primitives [Tennant81]. Thus in principle, there are no limitations to the applicability of our methodology due to the simplicity of our specification language.

For this thesis and for our implementation we use some straightforward notational conventions for algebraic specifications.

(1) There are two sections, labeled SYNTAX and SEMANTICS.

(2) All free variables in the semantics section are declared in the *Declare* line.

(3) All free variables in the syntax section are represented by their type.

(4) The IF THEN ELSE construct is capitalized.

(5) All operations are of the form $op\ (x^*)$ even if x^* is empty.

Figure 3.10 contains an example of a specification of the abstract data type *Queue Of Integers*, using our specification language. A large example of a specification of an ADT using our language can be found in Appendix III.

```
Type queue

SYNTAX
        newq()                  ->queue
        add(queue,integer)      ->queue
        remove(queue)           ->queue
        front(queue)            ->integer
        isnewq(queue)           ->boolean

SEMANTICS

        Declare q:queue,i:integer

1) isnewq(newq())               =true
2) isnewq(add(q,i))             =false
3) remove(newq())               =newq
4) remove(add(q,i))             =IF isnewq(q) THEN newq()
                                 ELSE add(remove(q),i)
5) front(newq)                  =-1
6) front(add(q,i))              =IF isnewq(q) THEN i
                                 ELSE front(q)

END.
```

Figure 3.10: Sample Specification Syntax

3.3. T-3

We have implemented the new testing methodology outlined in the previous section in a system called T-3 (Type Testing Tool). T-3 implements this new testing methodology by executing the high level system description given in Figure 3.11.

The procedure described in Figure 3.11 requires an object language which is the language in which the ADT is to be implemented. For T-3 we have chosen MODULA-2 as an object language. We choose MODULA-2 for two reasons:

(1) Its "modules" are a good implementation of abstract data types.

(2) It is readily available.

To "produce the next instance class" as required in step 3), we have based T-3 on the *Regularity Hypothesis* used by Bouge [Bouge85a, Bouge85b, Bouge86] as described in Section 2.1.1. We assume it is possible to associate a level of complexity with each member of the input domain of the software to be tested. For the T-3 system, as with previous work [Bouge85b, Bouge86, Choquet86], complexity is measured by the length of the trace that generates any particular instance class. T-3 will generate more and more complex (longer) instance classes until some termination condition is met. As outlined in steps 7.1) and 7.2) of the methodology, that condition is met when a failure is detected or a given number of test cases have been run.

1) Take as input the algebraic specification of the ADT
   and its implementation
2) Start a PROLOG interpreter with the PROLOG form of the
   ADT's specification as its program.
3) Produce the next instance class to be tested and
   put it in a form appropriate for input to
   the PROLOG interpreter.
4) Apply all O-type functions to the result of 3) and pass
   these to the PROLOG interpreter as goals.
   4.1) FOR each goal:
   4.2) Determine all possible solutions to each goal.
      4.2.1) FOR each solution
      4.2.2) Generate random values for the variables
         in these solutions so that they satisfy
         any constraints.
      4.2.3) Translate this into an object language
         test.
5) Append all the object language test code from 4)
   to driver code to form a main program module.
6) Compile and link this program with the ADT implementation
   modules and any library modules.
7) Execute this program
   7.1) IF an error is found: stop and report it.
   7.2) ELSE IF enough tests have been run: stop
   7.3) ELSE Goto step 3.

Figure 3.11: A New Testing Method

To implement step 3), T-3 generates the set I_n of instance classes to be tested by starting with I_0 containing the instance classes whose traces contain only the N-type functions. For the queue example, I_0 = { {Newq} }. T-3 then applies all the C-type functions that occur in the specification to the new members of I_n. The

resulting traces are added to the set I_n to produce a new set I_{n+1}. For the queue example:

$I_1 = \{ \{Newq\} \cup \{Addq((Newq),Integer)\} \}$
$I_2 = \{ I_1 \cup \{Addq(Addq((Newq),Integer),Integer)\} \}$

In step 4), test cases are produced by applying all the O-type functions to all traces in the set I_n. The resulting MODULA-2 code is of the same format as the code in Figure 3.7. The various steps of the PROLOG test procedure as outlined in Section 2.3.1 are incorporated into the parts of T-3 that implement steps 3), 4) and 7) in Figure 3.11.

Appendix II contains a detailed outline of the operation of T-3. This includes the main shell script (Listing A2.8) that can be repeatedly called to execute steps 4) through 7) of our method. Appendix II also includes listings of the significant routines and samples of the various files that are passed between routines. The reader is referred there for implementation details of T-3.

Figures 3.12a and 3.12b are block diagrams of the previous (DAISTS) system and T-3. As can be seen, T-3 is a significant improvement as it only requires two sources of information. Those are the specification and implementation. The software tester does not need to supply test cases and test data.

As shown in Appendix II, T-3 does in fact test appropriate software implementations. It did find errors in early versions of an implementation of our sample queue

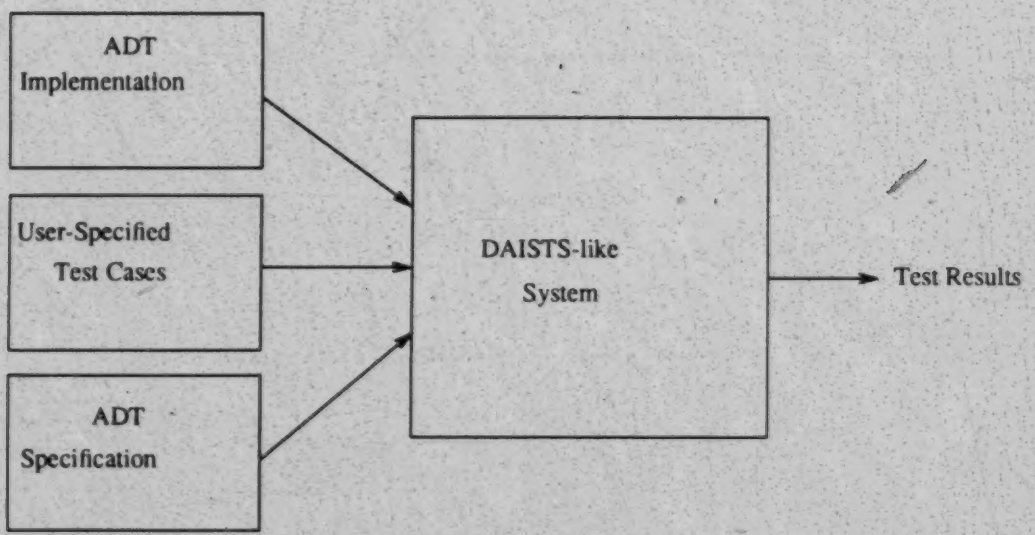


Figure 3.12a Previous Systems

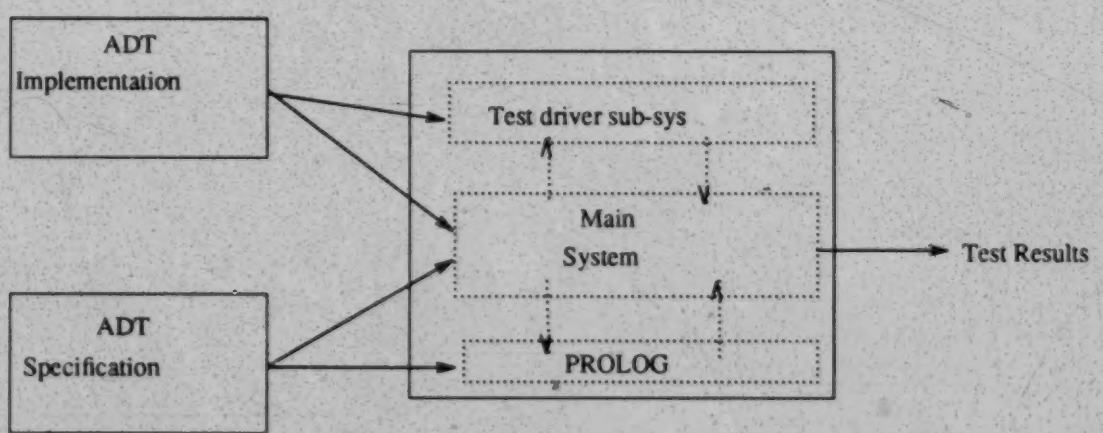


Figure 3.12b T-3 Testing Tool

type.

3.3.1. Improvements

T-3 represents a good first step in combining the methods using PROLOG to generate test cases for ADTs with automated harnesses such as DAISTS. T-3 has all the advantages of DAISTS as outlined in Section 3.1.6 and has solved some of the problems with DAISTS. In Section 3.1.7 we outlined two problems associated with testing with systems such as DAISTS. The first problem was that DAISTS required every function that is even mentioned in the specification to be implemented. This included hidden functions in the specification; T-3 solved this problem. By using the PROLOG interpreter as an oracle (step 4) Figure 3.11) as opposed to directly using the specification to determine the expected result, we do not need to call *hidden* functions of the ADT to determine what the specified result value should be. This is a significant improvement as the point of specifying an ADT is to separate implementation considerations from specification considerations. Requiring the implementation of certain internal functions prevents this separation of considerations and certainly violates the principle of information hiding.

The second problem with DAISTS was the requirement of an equality function for each new type. Our Queue example does not have an equality function defined on it (in this case there is no *equalq(Queue, Queue)* function), yet T-3 has no problem handling it (see Appendix II). We have solved this problem by changing the question

"Is Queue-A equal to Queue-B?" to "Are the results of all *Observations* I can make about Queue-A the same as the results of the *Observations* I can make about Queue-B?" For ADTs the second question reduces to "Are the results of the O-type functions the same?" Step 4) of our procedure (Figure 3.11) implements that query.

3.4. Problems

In building T-3 we discovered new research problems that needed to be solved. Some of these problems were solved during the process of building T-3 and are discussed in Section 3.3.1., whereas some still remain to be solved. In Section 2.3.5 we outlined three problems with the use of PROLOG to generate test cases. Those problems still exist in T-3 and need to be solved.

(1) There is a potential for infinite backtracking.

(2) Only equality predicates work can be used in the specifications.

(3) There is no theoretical basis for usefully ranking instance classes.

In Chapter 6 we outline a new method for test case generation which we have implemented in PROLOG. This methodology solves problems (1) and (2).

While we were developing a theoretical basis for ranking instance classes to be tested, we discovered a fourth problem which needed to be addressed before we could consider the problem of ranking instance classes. We found that

(4) the theoretical foundation for specification-directed testing of ADTs was not sufficient to build a usable practical system.

We found there were two theoretical sub-problems which needed to be addressed.

(4-a) We found that certain ADT and software testing assumptions are being implicitly applied without being explicitly stated.

(4-b) We also found that a reasonable goal for the specification-directed testing of ADTs has not been developed.

Chapters 4 and 5 give the results of our research into problems (4-b) and (4-a) respectively. Finally Chapters 7 and 8 present our solution to problem (3).

4. A THEORY FOR ABSTRACT DATA TYPE TESTING

In this chapter we present the foundations necessary for our testing methodology. We present what we have found to be a useful view of ADTs. That view allows us to characterize the ways an ADT implementation can fail. We then develop a goal for the specification-directed testing of ADTs based on a new view of the goal of the general software testing problem. We then give a theoretical justification for our method of addressing that goal and compare it with previous methods.

4.1. What is Being Tested

Guttag and others [Guttag77, Liskov74] have put forward the case that Abstract Data Types are very useful and powerful tools for programming. ADTs improve the ease of use and convenience of a programming language as well as facilitate the production of more reliable software.

As stated by Gougen [Gougen77]: "An algebra is simply a set, called the carrier of the algebra, together with an indexed family of operations defined on (cartesian powers of) that set. A many sorted algebra consists of an indexed family of sets called carriers and an indexed family of functions defined on the cartesian products of those sets." From a specification perspective, an ADT may be viewed as a many sorted algebra and "may involve several different sorts of things" such as truth values, integers, or stack states.

"All things of sort s are lumped together into a set, called the "carrier" of sort s." [Gougen78]

In terms of a testing context $<L,S,(\Pi),A>$, the indexed family of sets (S) and the indexed family of operations (V) of the many sorted algebra described by the ADT, implement the L(S) language of the testing context.

4.2. Abstract Machine

As with any data type, an *abstract* data type is basically a set of values and a set of operations on those values. The adjective "abstract" in "abstract data type" is generally accepted to refer to an independence of representation. To examine the problem of how an ADT implementation can fail and how independence of representation affects testing, we have found it useful to use Liskov and Zilles [Liskov74] definition that "an abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects."

Abstract data types are useful to programmers because they allow the programmer to be concerned only with the behavior which an (abstract) object exhibits, and relieve the programmer of the concern of how that behavior is achieved. In this sense ADTs are similar to the "standard" types provided by most programming languages. A programmer does not normally care how an integer is represented internally in a machine, or what series of machine operations are

required to perform a particular integer operation. That programmer is only concerned with the fact that 2 ** 4 =16. That is, the programmer is normally only interested in the behavior of the integers, not how they are implemented. When a programmer uses a "standard" data type he is making use of an abstraction which is implemented at a lower level. That level could be the compiler level or lower. An ADT can also be used at one level and implemented at another. The difference is that for an ADT, the lower level does not come into existence simply by being part of the language; rather an ADT exists through a cluster of programs that implement the operations which define that ADT.

We may view ADTs as hierarchical in nature. They may be built on top of each other without knowing how the lower types are implemented. We do not need to know whether lower types are implemented in silicon, in the complier, or as an ADT. In any of those cases we can view the ADT we are building and testing as running on top of an abstract machine. This abstract machine provides all the types and operations necessary to conveniently specify and build the ADT [Liskov74].

Since an ADT is built to run on an abstract machine, we can view the testing of ADTs as something that we apply recursively to a set of embedded systems. In such a system each object is made up of objects below it. For an implementation of an ADT to fail in such an embedded system one of three things must happen:

(1) one of its lower components fails;

(2) the implementation's use of a component does not match that lower component's specification;

(3) the way in which the implementation combines the results from its component parts does not match the ADT's specification.

To handle (1) above, specification-directed testing of ADTs should be performed in a bottom-up manner. In that way we may assume that the abstract machine executing the ADT is correct. Thus we may assume, *for test set generation purposes*, that case (1) does not occur. We are then concerned with failures (2) and (3) above which deal with whether the implementations of the functions of our ADT are correct.

4.3. Goals of the Testing Method

In Chapter 2 we introduced the predicate OK over the input domain D of a program, where OK(d) means that the implementation behaves correctly for d ∈ D. The correctness of an implementation can therefore be expressed as "$\forall d \in D \;\; OK(d)$."

We have found it useful to view the predicate OK as a conjunction of two other predicates OK_b and OK_w: These terms are short for "OK-black-box" and "OK-white-box." In general terms, OK_b implies that the implementation does everything it is supposed to do. OK_w implies the implementation only does what it is supposed to do.

Thus we can say:

$$OK(d) \Leftrightarrow OK_b(d) \wedge OK_w(d).$$

A computational state can be viewed as having two components: "the environment, which is used as a record of identifier bindings, and the store, which is used as a record of the effects of assignments" [Tennent81]. Consider a program F whose input domain is the set of data D. The result of executing F with input $d \in D$ may be viewed as a 6-uple $<d, o, s_i, s_o, e_i, e_o>$ where d is the input value, o is the output value, s_i and e_i are the store and environment before execution, s_o and e_o are the store and environment after execution. We will denote this 6-uple as Result(F(d)). We will use Out(d,F(d)) to mean that the output values from F given d do not violate F's specification.

Definition (OK_w and OK_b):

> Given a function F with domain D, the result of executing F with input $d \in D$, $Result(F(d)) = <d, o, s_i, s_o, e_i, e_o>$ and an expected result $Result(F(d))^* = <d, o^*, s_i, s_o^*, e_i, e_o^*>$
>
> THEN $OK_b(d) \rightarrow \{o = o^*\}$
> $OK_w(d) \rightarrow \{(s_o = s_o^*) \wedge (e_o = e_o^*)\}$

Now the testing process can be viewed as two separate tasks:

(1) determining T_b such that $[\forall (t) \in T_b \ \ Out(t, F(t))] \rightarrow [\forall d \in D \ \ OK_b(d)]$; and

(2) determining T_{ω} such that $[\forall(t)\in T_{\omega}\ \ Out(t,F(t))] \rightarrow [\forall d\in D\ \ OK_{\omega}(d)]$.

We separate OK into white box and black box cases because white box testing alone cannot say (without further information) that the implementation "does everything it should." To make that statement we need to know everything the system is specified to do. That in turn requires black box methods for analyzing the specifications of that software. Similarly, specification-directed black box testing cannot say (without further information) that the implementation "does only what it is supposed to." To make that statement we need to determine all possible results for all possible inputs. That in turn requires white box methods for analyzing the implementing code.

Since we are working on a "Specification Directed Software Testing Method," the best we can do is try to ascertain that the implementation does everything it should. In the remainder of this thesis we will assume that our overall software testing goal is *to find a test element for which the implementation does not do everything it should*. That is, our overall testing goal is to make the system under test fail.

For our purposes a correct function is a function that is consistent with every axiom of its specification for all possible inputs. That is:

Definition (Correct Function):

Let A be an arbitrary function, and
Let D(A) be the domain of the function A

Then we define the predicate "Correct" as:

$$Correct(A) \rightarrow \forall x \in D(A)(OK_b(A(x)))$$

As we outlined in Chapter 2, all the observable functionality of an ADT is supplied by its O-type functions. Therefore we now define a correct ADT implementation for our purposes as follows:

Definition (Correct ADT Implementation):

For an ADT $T=(V,S)$
And an implementation I of T

$$Correct(I) \rightarrow \forall O\, I(O \in V \wedge O \in O\text{-}type)(\forall x \in D(O)(OK_b(O(x))))$$

Where $O\text{-}type$ is the set of all O-type functions in T.

It is important to note that our new definition of a correct ADT implementation is independent of the TOI of that ADT. This in turn will allow us to develop a testing method that does not ask the unanswerable question "is this element of the TOI correct ?" This is a critical advance in our method. The implementation of the TOI will vary from programmer to programmer, therefore a testing criterion that answers the question "is this element of the TOI correct?" cannot be built without understanding how the ADT was implemented (white box testing).

Given this definition of a correct ADT implementation and our overall goal for software testing we can state that the operational goal of our software testing methodology is *to detect failures in a software system.* A failure is an observable event where a system is inconsistent with its specifications. A failure should not be confused with an error which is a piece of information (code or data) which when processed by the system may produce a failure. Thus our goal is to demonstrably show that

$$\exists O \mid O \in V \wedge O \in O\text{-}type\ (\exists x \in D(O))(\overline{OK_b}(O(x))))$$

Either an implementation is correct or it is not. If it is correct it does not matter which functions we test, the results of our tests will match the specifications. If the implementation is incorrect, then there exists a set of test inputs that will, at some point, cause an observably incorrect result to be produced. Since O-type functions are the only functions to produce an observable result, we need only concern ourselves with analyzing the output for them. Since the choice of test cases will make no difference to a correct implementation, we are justified in assuming the implementation we are testing has a fault. We then try to make the system fail, and to make the system fail as quickly as possible using as few resources as possible.

We would also like, in a more general sense, the continued testing of a software system to increase our confidence in that system. With infinite resources, extrapolating testing to infinity should yield a proof of correctness or at least a proof of no

incorrectness. That is:

$$\lim_{Tests(ADT)\to\infty} \rightarrow NOT(NOT(Correct(ADT))).$$

The key point here is that we are aiming for a hierarchy of tests; one test to be applied after the other such that the more tests we execute the more likely it is that the software is not incorrect. Note we are not saying "the more tests we (successfully) run the more likely the software is correct." That approach is not practical because a test cannot show the absence of errors, only their presence.

4.4. A General Testing Method

Using the definitions of the previous sections, our testing method (M) uses some assumptions (A) about the system under inspection, as well as an implementation (I) of that system and an algebraic specification (S) of that system, to produce a test case series. It then calls for the execution of the associated test series in such a way that the more ADT operations we execute, the more likely we are to detect a failure. Figure 4.1 outlines the general function of any testing method. Note that the result of testing is dependent on four separate entities: the assumptions (A), the implementation (I), the specification (S), and the testing method (M).

It is impractical to require a user to repeatedly state his assumptions about a system every time he wants to test it. Indeed the very nature of any testing

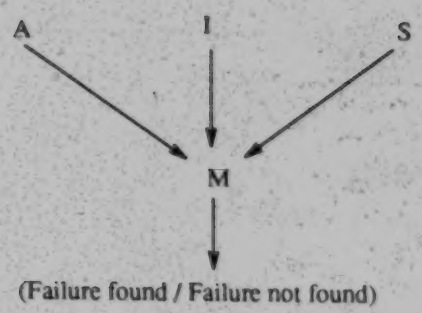


Figure 4.1: The General Testing Function

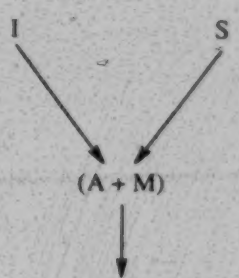


Figure 4.2: An Operational Outline of a Testing Method

method is predicated on assumptions about the nature of the system to be tested. Thus for any testing method there are some assumptions about the system under test that should not be separated from the testing method. Figure 4.2. gives an operationally useful outline of the function of our testing method. Note how Figure 4.2 matches Figure A2.1 (flow chart of the operation of T-3), but that the assumptions (A) are bundled with the method (M) and the two are not independent. Therefore a description of a testing method is incomplete without a description of the assumptions attached to it.

In Chapter 5 will we outline the assumptions (A) of our methodology.

4.5. The Testing Method

Algebraic specifications of ADTs do not involve any existential quantifiers. Bouge [Bouge85a] shows that this ensures the existence of an acceptable collection of test instance classes, given the extra hypotheses of uniformity and regularity as discussed in Section 2.1. Our uniformity hypothesis will be defined and justified in Section 6.1.

For the regularity hypothesis we assume it is possible to associate a level of complexity with each member of some input subdomain of the program under test. Complexity was discussed in Section 2.1.1. A regularity hypothesis states that the program behaves "regularly" with respect to this measure. A program behaves regu-

larly with respect to a measure if the fact that the implementation works correctly for all input of complexity less than some level k in the domain implies that it works correctly for any input in that domain. The following is our regularity hypothesis.

Regularity Hypothesis:

There exists a k such that

$[\forall d \in D\ (complexity(d) \leq k \rightarrow OK(d))] \rightarrow \forall d \in D\ \ OK(d)$

Where D is some subset of the input domain of the implementation.

We observe that this hypothesis is trivially satisfied as k tends to infinity.

If "$complexity(d)$" can be calculated for all elements of the input domain of the ADT, then an *acceptable* (as per definition in Section 2.1.1) battery of tests will be $<H,(T_n)_{n \in N}>$ where (T_n) is a family of tests $T_1, T_2, T_3, \ldots$. For any particular subscript k, passing the set of experiments (also called a test) T_k will imply, in combination with the assumptions H about the testing method, that all the elements of the input domain with complexity less than or equal to k are "OK." The assumptions about our method are discussed in Chapter 5 and our uniformity hypothesis is described and justified in Section 6.1. Here we are concerned with describing the properties of a complexity metric that will allow us to produce a test set T_k that will check if all the elements of the input domain with complexity less than or equal to k are OK. Thus we must view each test T_k in the family of tests $(T_n)_{n \in N}$ as follows:

$$T_k = (\forall d \mid (d \in D \wedge complexity(d) \leq k) OK(d)?)$$

Since the subscript in (T_n) is only defined on the natural numbers, and therefore has a lower bound, there must be a lower bound for the complexity of any element of the input domain. Normally that lower bound will be zero. To be able to produce a test T_i we must be able to calculate a complexity value for all elements of the input domain. Thus we define a valid complexity metric as follows:

Definition:

We call M a valid complexity metric for the inputs I_p to program P;

IF:

(1) There is a value min for which there does not exist an input I_p of program P for which $M(I_p)$ is less than min.
That is, $(\forall I_p,\ M(I_p) \geq \text{min})$

(2) $M(I_p)$ can be finitely calculated for all possible inputs I_p for program P.

The commonly used complexity metric [Wild86, Bouge86, Choquet86, Bouge85b] for an instance i is the number of ADT-function calls in i. We note that such a complexity metric satisfies our definition of "a valid complexity metric."

4.5.1. Previous Methods

As we have stated, the length of an element of L(S), for a testing context $\langle L, S, (\Pi), A \rangle$, is the complexity metric that has been used [Wild86, Bouge86, Choquet86, Bouge85b] to perform step B of the Testing Process Diagram (Figure 1.1). That step produces batteries of tests for that context. We agree that such a metric does

guarantee an *acceptable* battery of tests as it is a valid complexity metric as defined above. A major problem with these previous methods is how this metric is applied. We quote from Bouge [Bouge85b] where "X" is any ADT:

> "Consider the case where a set of constructors" {N and C-type functions} "is given together with the specification of the type of interest. Hypotheses can be strengthened by assuming that X is actually finitely generated with respect to those constructors. Instantiation may thus be limited to those terms of size less than k which are combinations of constructors. The number of generated instantiations is then considerably decreased. This corresponds precisely to optimizing a test set by discarding redundant tests. This optimization is usually left implicit in testing methodologies."

This statement was a significant step forward in that it at least recognized that "This optimization is usually left implicit in testing methodologies."

We examine the statement "This corresponds precisely to optimizing a test set by discarding redundant tests." We will use for our examples the ADT "Queue with Has" as specified in Chapter 2. The axioms of that specification are:

(1) Isnewq(Newq)	-True
(2) Isnewq(Addq(q,i))	-False
(3) Deleteq(Newq)	-Newq
(4) Deleteq(Addq(q,i))	-If Isnewq(q) THEN Newq ELSE Addq(Deleteq(q),i)
(5) Frontq(Newq)	-error
(6) Frontq(Addq(q,i))	-IF Isnewq(q) THEN i ELSE Frontq(q)
(7) Has(Newq,i)	-False
(8) Has(Addq(q,i),j)	-IF i=j THEN True ELSE Has(q,j)

Consider that we have a test case

$t_a = o_1(e_1(f_a(x)))$

Where o_1 is an O-type function.

e_1 is an E-type function.

Since e_1 is an E-type function then there is a C-type function $\underline{c_a}$ such that according to the specifications

$e_1(f_a(x)) = c_a(f_b(y))$

Where c_a is a C-type or N-type function

$f_b(y)$ contains only C-type and N-type functions

According to the optimization as described by Bouge if we have test cases:

$t_a = o_1(e_1(f_a(x)))$

$t_b = o_1(c_a(f_b(y)))$

we need only run the test "t_b" as the test "t_a" is a "redundant" test.

For our Queue example if we have the tests:

t_a=*Frontq* (*Deleteq* (*Addq* (*Addq* (*Newq* ,1),2)))
t_b=*Frontq* (*Addq* (*Newq* ,2))

By applying axiom (4) to t_a:

Frontq (*Deleteq* (*Addq* (*Addq* (*Newq* ,1),2)))=
Frontq (*Addq* (*Deleteq* (*Addq* (*Newq* ,1)),2))

By again applying axiom (4):

Frontq (*Addq* (*Deleteq* (*Addq* (*Newq* ,1)),2))=
Frontq (*Addq* (*Newq* ,2))

Thus:

t_a = *Frontq* (*Addq* (*Newq* ,2)) = t_b

Therefore, by their "optimization," we need only run the test:

Frontq (*Addq* (*Newq* ,2))

because once we have done that test, the test

Frontq (*Deleteq* (*Addq* (*Addq* (*Newq* ,1),2)))

becomes redundant.

This optimization is necessary to produce test sets that can be run with finite resources. Without the optimization an ADT with n functions would require greater than n^k tests for a maximum trace length of k. Clearly the number of tests would be

large for even a small upper bound on instance length.

We found in running T-3 with test batteries based on this method of test set generation, there were some faults in some implementations that were never exercised. On inspection we found that even as k tended to infinity and we tested forever, these faults would not be exercised. According to Bouge's theory [Bouge85a], this is not possible. Figure 4.3 contains an example of such a fault. The queue is implemented with a standard double linked list using the following MODULA-2/PASCAL record type:

```
queue_element =     record
                    value:      Integer
                    previous:   ^queue_element
                    next:       ^queue_element
                end

queue =         record
                top:    ^queue_element
                bottom:         ^queue_element
            end
```

When "Deleteq" removes an element it simply moves the "top" pointer to the next element. The "Has" function starts at the bottom of the queue and scans until it finds the element it is looking for or it encounters the NIL element. The "Has" function may scan past the top of the queue. Since elements past the top of the queue have been deleted and should not be considered, this is an algorithmic defect (a fault). This is a very simple fault spread over two functions.

```
PROCEDURE deleteq (q:queue): queue;
BEGIN
        IF q.top  = NIL
                THEN RETURN q
                ELSE
                        q.top = q.top^.next;
                        RETURN q
        END;
END Deleteq;


PROCEDURE has (q:queue , term:INTEGER): BOOLEAN;
        VAR current : qpointer;
BEGIN
        current := q.bottom;
        WHILE current <> NIL DO
                IF current^.value = term
                        THEN
                                RETURN TRUE
                        ELSE
                                current := current^.previous
                END;
        END;
        RETURN FALSE
END has;
```

Figure 4.3: Code with a Fault over Two Functions

For this example the fault would be detected by the test:

Has(Deleteq(Addq(Newq,1)),1)

The correct result is "false" and this implementation returns "true." This simple logic fault is never exercised because by applying axiom (4) we can say:

Has (*Deleteq* (*Addq* (*Newq* ,1)),)=
Has (*Newq* ,1)

Thus according to the optimization as described by Bouge,

Has (*Deleteq* (*Addq* (*Newq* ,1)),1)

becomes redundant once the test:

Has (*Newq* ,1)

has been run. Therefore the test:

Has (*Deleteq* (*Addq* (*Newq* ,1)),1)

is never run nor is *any* test of the form:

Has (*Deleteq* (*Addq* (*F* (*X*))),*a*).

This means that this fault is never discovered even as the complexity tends to infinity. As we have found a simple counter-example, there must be something wrong with the previous test selection theories of Bouge and others [Wild86, Bouge86, Choquet86, Bouge85b].

In re-analyzing this method of generating a battery of tests we found that as long as a complexity metric satisfies our new definition of a *valid* complexity metric, it will produce an *acceptable* battery of tests. The complexity metric "length of trace" is such a *valid* metric and does produce an acceptable battery of tests. We have found the problem is in the second last sentence of Bouge's statement that we quoted: "This corresponds precisely to optimizing a test set by discarding redundant tests." This is, prima facie, a reasonable statement. According to the axioms we can certainly say (by simply applying axiom (4) once):

$$Has(Deleteq(Addq(Newq,1)),1) = Has(Newq,1).$$

The problem here is: we are testing the implementation, not just using it. We must assume there is an fault in the implementation. Therefore we cannot assume $Has(Deleteq(Addq(Newq,1)),1)$ will be implemented properly. If

$$Has(Newq,1)$$

and

$$Has(Deleteq(Addq(Newq,1)),1)$$

are both implemented properly then

$$Has(Deleteq(Addq(Newq,1)),1) = Has(Newq,1).$$

As software testers we cannot assume this equality. Thus the optimization "usually left implicit in previous testing methodologies" does NOT correspond precisely to optimizing a test set by discarding redundant tests.

This optimization corresponds to adding the assumption "The implementation of all E-type functions is correct" as one of the assumptions in (A+M) in Figure 4.2. As shown in Section 4.4 we cannot "unbundle" the assumptions of a methodology from the methodology. In terms of Figure 4.2, we cannot separate "A" from "(A+M)". Therefore, to use those previous methods [Wild86, Bouge86, Choquet86, Bouge85b], we must assume a potentially large part of the implementation is correct. Consequently, those previous methodologies do not produce an acceptable battery of tests.

4.5.2. Our Method

At the start of Section 4.5 we showed that to test an implementation of an ADT we produce a battery of tests:

$$(T_k)=T_1,T_2,T_3,...$$

such that each T_k asks the question:

$$\forall d \mid (d \in D \wedge complexity(d) \leq k) OK(d)?$$

for a valid complexity metric. That is, we produce a battery of tests T_k that, in combination with assumptions about our method, shows that all elements of the input domain with complexity less than or equal to k are "OK". We know that:

> "A testing methodology that iteratively tests instance classes in an order dictated by what we call a valid complexity metric will produce an acceptable collection of tests."

Therefore at the highest level our testing methodology is:

Iteratively test instance classes of the ADT in the order dictated by our complexity metric.

We know this will produce an acceptable battery of tests as long as our complexity metric is valid.

We could not find a complexity metric based solely on the syntax of the ADT that was both "valid" and practical to use. We have developed a valid metric based on a new model of the computations of an ADT. That metric will be presented in Chapter 8, after we have developed a usable way of testing instance classes in Chapter 6 and presented our model for the computations of an ADT in Chapter 7.

5. A FOUNDATION FOR AUTOMATABLE SPECIFICATION-DIRECTED TESTING METHODS

In the previous chapter we showed how dangerous it can be not to explicitly state the assumptions intrinsic to a software testing methodology. In this section we will describe the universe in which our specification-directed software testing methodology is to work. We believe that it is important to state as explicitly as possible what our methodology is meant to do. We have searched the literature on specification-directed test harnesses [Gannon81, McMullin83] and previous work on test case generation using PROLOG [Wild86, Bouge85b, Bouge86, Choquet86]. We have found that a common shortcoming of this previous work is that very little or no information was given defining and describing the domain of software to which the results are applicable. Therefore we have undertaken to investigate and specifically state, as reasonably as possible, the assumptions and constraints on the domain of software to be tested by our specification-directed software testing methodology. Our results are based on an implementation of our methodology. We believe this is an important step forward, not only for specification-directed test harnesses and test case generation using PROLOG, but also for specification-directed testing in general. We leave for future work the investigation of the impact of loosening the assumptions and constraints developed here.

5.1. Software Testing Assumptions

We found it necessary to make some "standard" software testing assumptions about the software we are going to test. The first of these assumptions concerns "coincidental correctness." In software testing, coincidental correctness occurs when a fault is tested and yet coincidentally the test data results in correct output variables [White87]. If different test data had exercised that fault, incorrect output variables would have resulted. A common software testing assumption is the "no coincidental correctness" assumption which assumes that this does not happen for any reason.

Coincidental correctness occurs when a fault is tested yet, coincidentally, the test data results in correct output values [White87]. We make a less restrictive assumption than no coincidental correctness. We assume that two incorrect results do not cancel each other out. That is, we allow that for any *experiment* E, the implementation may return an incorrect result for a function call in E. We assume that a second function call in E will not return another incorrect result that, in combination with the first incorrect result, will cause a correct overall result for E. This assumption is called the "no coincidental incorrectness" assumption and is not as restrictive as the coincidental correctness assumption because it allows the existence of *some* faults that are exercised with no output manifestation.

To see how "no coincidental correctness" and "no coincidental incorrectness" differ, let us take two functions specified as follows:

$f_a(A,B)=A+B$
$f_b(C,D)=C^D$.

Let us assume there is a fault in f_a so that instead of returning the value $A+B$ it returns the value $A-B$.

If we run a test:

$$Output = f_a(7,-5) + f_a(32,5)$$

we will get the output value 39 which is the correct result. In this test the error in the first f_a calculation is canceled out by the error in the second f_a calculation. A "no coincidental correctness" assumption would assume this does not happen. A "no coincidental incorrectness" assumption would also assume that this does not happen.

If we run a test:

$$Output = f_b(1, f_a(5,4))$$

we will get the correct output of 1 because 1 raised to any power is 1. Again we have tested the fault and coincidentally the test gives the correct output values. If we say we are assuming "no coincidental correctness" then we are assuming this does not happen. With "no coincidental incorrectness" we do not make that assumption because in this case the error is not masked by another error. No coincidental incorrectness only assumes that an execution of a fault is not masked by another execution of a fault.

We assume that the ADT is testable. To do that we assume that for each operation o associated with an abstract data type, there is a set of operations O_o with the following properties:

(1) o^* is in O_o , where o^* is the correct operation.

(2) o is in O_o , where o is the implemented operation.

(3) There is a method of selecting tests for o such that for any o' in O_o, if o and o' agree on these tests, then they are equivalent.

These three assumptions are necessary for the ADT to be testable [Howden85]. Our regularity hypothesis (Section 4.5)

$$[\forall d \in D\ (complexity(d) \leq k \rightarrow OK(d))] \rightarrow \forall d \in D\ \ OK(d)$$

follows directly from property (3). If a failure can happen and there is a test $t_{failure}$ that can reveal that failure, then our regularity hypothesis holds when we set our complexity limit k tc the complexity of the test that reveals the failure:

$$k = complexity(t_{failure}).$$

We have found the "no coincidental incorrectness," and "testable" assumptions necessary for a workable system.

5.2. Abstract Data Type Assumptions

The most important assumption we make about the software to be tested is that it is in the form of an abstract data type. That is, we are testing a set of

operations that together define a set of abstract objects (the elements of the abstract data type). These operations are the only operations permitted to manipulate those objects directly. Thus for any abstract data type *T* we can assume the existence of a language L(T) which is defined by that abstract data type. For testing purposes this means *we can assume for a testing context* $\langle L(S), S, (\Pi), A \rangle$, *L(S) does exist and is a language defined by the ADT to be tested.* Facilities for producing modules that implement abstract data types are available in such programming languages as SIMULA, CLU, MODULA-2, Concurrent EUCLID, and TURING.

We have found it necessary to assume for our testing methodology that the type T= [{V}, {S+O}] is restricted to only allow total functions as members of {S+O}. We are not saying that nonsensical inputs or inputs that may cause error conditions are not acceptable; rather we are only requiring that the operations never fail to terminate and are defined over all of their domain. We grant that some software, such as an operating system, is non-terminating, but in most cases programs are meant to terminate. We agree with Guttag and Horning's [Guttag78a] argument that it is hard to imagine a useful type T= [{V}, {S+O}] where O contains a potentially non-terminating operation. We justify this assumption as reasonable by observing that: testing abstract data types without making some assumptions about their termination would involve solving the

halting problem.

In Section 4.2 we showed that ADTs are hierarchical in nature and that they are meant to run on an "abstract machine." By following a bottom-up test plan we showed that we can assume that this abstract machine works correctly. The abstract machine provides all the types necessary to build the ADT [Liskov74]. This means that for the representation phase of the testing process (Step A, Figure 1.1), we can assume that the set S in the testing context $\langle L(S), S, (\Pi), A \rangle$ is correct.

We have found that for our testing methodology and for previous methodologies [Bouge85b, Bouge86, Choquet86, Wild86], it is necessary to assume that the functions to be implemented by the abstract data type to be tested are primitive recursive. We recall that a primitive recursive function is any function that can be obtained from certain initial functions by a finite number of applications of composition and recursion. In the simplest case these initial functions can be the zero function, the successor function and projection functions. While some functions are not primitive recursive, most of the useful computable functions are primitive recursive. Thus we do not believe this significantly limits the applicability of our methodology. Our main reason for assuming primitive recursive operations is that we will be requiring an algebraic specification to exist (see following section). This in turn requires an axiomatization of a containing algebra[9]

[9]Containing Algebra: "An algebra that, by the forgetting of some operations, can be restricted to the intended algebra" [Guttag78a].

for our type T= [{V}, {S+O}] [Guttag78a]. Requiring primitive recursive operations ensures the existence of such an axiomatization [Guttag78a]. The existence of such an axiomatization has always been assumed in previous research in using PROLOG to aid in test set generation. Since we rely on such an axiomatization, we must BE SURE that it exists, and know what constraints that imposes.

A useful side effect of assuming only primitive recursive operations in {S+O} for a type T=[{V}, {S+O}] is that all the elements of that type are finitely generated. This is significant because if it were not true there would be values of our ADT that we could not test.

5.3. Algebraic Specifications

Our testing methodology requires the existence of two sources of information: the implementation of the ADT to be tested and an algebraic specification of that abstract data type (see Figure A2.1). In this section we will outline what we mean by "an algebraic specification."

We have found it necessary to make the following assumptions about the specifications of the abstract data type to be tested. Assumption (2) has already been discussed in Section 3.2.1. Each of the others will be discussed in the following sub-sections.

(1) They exist; not all software systems have formal specifications.

(2) They are algebraic and of the general form outlined in Section 3.2.1.

(3) They are correct; clearly, if the specifications are incorrect then testing that a system does not violate its specifications is not testing for correctness of the implementation.

(4) The axioms are consistent; if the axioms are not consistent then we may not be able to uniquely determine the correct output for a test.

(5) The axiomatization is sufficiently complete; this assumption assures us that we can determine at least one correct output for each test.

We have found assumptions (1), (3), (4) and (5) are also necessary, although unstated, for previous methods to work. Those methods were outlined in Sections 2.3 and 3.1.

5.3.1. Existence

That the specifications exist and that they are algebraic are the two basic premises of our work. Assuming the existence of algebraic specifications, while necessary for our methodology, is not always reasonable, and must be taken into account when considering the use of our testing methodology. If the software to be tested was developed using a rapid prototyping methodology it is probable that a formal set of specifications was never developed. The production of algebraic specifications for such a system may represent a significant cost which may be

prohibitive in some cases. One point in the defense of the cost of building such specifications is given in [McMullin83]. They found that requiring such specifications forced the consideration of conditions missed in even the best informal specifications. We also found that building the algebraic specifications for a List type forced the considerations of conditions that had been missed. We used as a basis for that type a two page verbal description given as an assignment to third year undergraduates at the University of Alberta. When we built the algebraic specifications we found that the two page verbal description did not consider all conditions. We refer to this "extended List" type as the List-e type, specified in Appendix III. This type contains two unusual functions for a list: intersection and union. The undergraduate assignment defined these functions in the usual set-theoretic way. That definition does not specify the order of the elements of the list that results from intersection or union. Therefore the result of applying an output function such as "getElt" to a list that is the output from a union or intersection was undefined in the undergraduate assignment. Note that it is defined in our specification.

5.3.2. Correctness

The problem of determining the correctness of the specifications of a unit of software is beyond the scope of this research. The purpose of our work has been to investigate specification-directed software testing. As such we are concerned with showing that an implementation and a specification are inconsistent. For

a software module to be "correct," it is sufficient that it be consistent with its specifications ($\forall$ d $\in$ D OK(d)), and that the specifications are correct. The issue of correctness of specifications deals the problem of specifying systems that cannot be realized, or that are not usable. This may happen if the specifications do not provide the functionality the user desired. The reader is referred to [Kemmerer85] for an investigation into the determination of the correctness of a specification. For our work we assume the specifications we are given are correct in that they provide the desired functionality.

5.3.3. Consistent

A major issue concerning algebraic specifications "is adequacy which comprises consistency and completeness" [Berztiss83]. The classical notion of consistency is that a theory is consistent if and only if it is impossible to derive a contradiction as one of its consequences. The classical form of a contradiction is

$$P \wedge \neg P$$

where P is any predicate. The traditional notion of completeness is that for any predicate P, either P or $\neg$P should be the consequence of the theory.

In building T-3 we have found that for the previous methods [Bouge85b, Bouge86, Choquet86, Wild86] and our methodology to work, we need to assume that

the specifications are consistent[10]. In an algebraic specification of an abstract data type the partial semantics of the type is supplied by the axioms in the semantics section. These axioms may be viewed as individual statements of fact. If two or more of these are contradictory, the axiomatization, and therefore the specification, is inconsistent.

Proving the consistency of an arbitrary set of axioms is, in general, an unsolvable problem. We observe that Guttag [Guttag80] found that the production of algebraic specifications of types with more than four constructor functions quite difficult to accomplish directly, but that it could readily be accomplished by breaking the type into several parts. Therefore, although the problem of assuring the consistency of very large axiomatizations is very difficult, it can be successfully attacked. We also observe that useful and consistent specifications have been produced for smaller types with less than four constructor operations. Therefore, the consistency of an axiomatization is reasonable to assume as a basis for an operational software verification methodology, even though it is unprovable.

5.3.4. Sufficiently Complete

In building T-3 we found that it is absolutely necessary to assume that the algebraic specifications are *sufficiently complete*. A *complete* axiom set is one to which

[10] "If we can use the statements to derive an equation that contradicts the axioms of one of the underlying types used in the specification, the axioms of the specification are inconsistent. Ultimately, any inconsistent axiomatization is characterized by the fact that it can be used to derive the equation true=false." [Guttag80]

an independent axiom cannot be added, or "one with which every well-formed formula or its negation can be proved as a theorem" [Guttag78a]. For abstract data types this is revised slightly to the property of *sufficiently complete*. The idea is that if L(T), which is the language defined by the abstract type T=[{V}, {S+O}], is sufficiently complete, then every term in this language must be assigned a meaning by the axiomatization.

Definition (Sufficiently Complete):

> "For an abstract type T= [{V}, {S+O}] and an axiom set A, A is a sufficiently complete axiomatization of T if and only if for every word of the form $Fj{\cdot}n(x1,\ldots,xn)$ contained in L(T) where $Fj{\cdot}n \in O$, there exists a theorem derivable from A of the form $Fj{\cdot}n(x1,\ldots,xn)=u$, where $u \in V_i \in V$." [Guttag78a]

Note that by this definition the axiomatization A is *consistent* if for each $Fj{\cdot}n(x1,\ldots,xn)$, u is unique.

As with consistency, the problem of ensuring that a set of axioms is sufficiently complete is, in general, undecidable [Liskov74, Guttag78a, Gougen78]. It has been shown however [Thatcher82], that if we limit the kinds of algebras in our domain and limit the language used to specify the axioms, then it is possible to produce a sufficiently complete axiomatization. These limits do not put any new constraints on the ADTs we will be testing. Therefore the consistency of the specifications is not an unreasonable or burdensome assumption.

We recall Guttag and Horning's [Guttag78a] theorem that states for any type T=[{V}, {S+O}], all of whose operations are primitive recursive, there exists an axiomatization *A* which is sufficiently complete. What was actually shown was that those conditions are sufficient to guarantee that for any term $o(x,y^*)$, $o \in O$ there exists a series of reductions:

$$o(x,y^*) \rightarrow Z_1 \rightarrow Z_2 \rightarrow Z_3 \rightarrow \cdots \rightarrow Z_n$$
$$\text{where } Z_n \in V_i,$$
$$\text{and } V_i \in V.$$

That is, for any term there exists a series of reductions or replacements that produce a non-TOI value. Every axiom in the semantics section of an algebraic specification can be viewed as a statement saying "the value or expression on the right side of this axiom can be used in place of the expression on the left side, whenever one is trying to evaluate an expression that contains the expression on the left side of the axiom." Therefore, any sufficiently complete axiomatization in that form may be viewed as a set of replacement rules. Being able to view axioms as replacement rules will be very useful to us as the PROLOG interpreter in our methodology treats the horn clauses we derive from those axioms as replacement rules.

5.3.5. Other Specification Techniques

Our testing methodology generates tests based on the algebraic specification of the ADT. Currently, there are two other basic families of specification tech-

niques available: state machine specifications and abstract model specifications. All three approaches "define behavior in units called functions, and do so in a result-oriented or non-procedural way that suppresses most detail of the implementation" [Berg82].

State-machine specification resembles other specification techniques in that it defines a set of functions that specify transformations on inputs. The set of functions may be viewed as defining the nature of an abstract data type or describing the behavior of an abstract machine. It divides the interface procedures into two classes.

V-functions: (for Variable or Value) report values. They have no side effects.
O-functions: (for Operations) change the state of the module.
They do not return values.

Specification languages based on this technique include SPECIAL, which was developed at SRI international, and INA JO which is part of a verification system at the Systems Development Corporation [Berg82].

The practical use of the state machine method is largely a result of its suitability for validation of security [Berg82]. In general practice, the state-machine approach has been found to be unwieldy and "unsatisfactory for more complex and arbitrary structures and for expressing effects that involve the evaluation of algebraic formulas" [Lamb88]. These limitations make current state-machine specification methods impractical for our specification directed testing methodology.

With the model oriented approach to specification, the concrete data structures that are manipulated by the programs are modeled via abstract mathematical objects such as sets and sequences. The meaning of the interface procedures is described via predicates on these abstract objects [Lamb88]. Once a concrete representation of the abstract model has been developed, a correspondence between the abstract and concrete representations is established via

(1) an abstraction function which maps a concrete data structure into an abstract model of the data structure, and

(2) a representation function, which maps an instance of the abstract model into the set of all possible concrete representations of the abstraction. This is a one-to-many mapping because there can be several representations of the "same" abstract value.

This approach was developed by Hoare and is very compatible with the programming languages Alphard and Euclid [Berg82].

There are two problems with using model oriented specifications as a basis for our testing methodology. The first problem is that model oriented techniques are not always universally quantified. Since all clauses in PROLOG are universally quantified this will make the use of PROLOG, at the same time as model oriented specification techniques, impractical.

The second problem with using model oriented specifications as a basis for our testing methodology is that they use operations on "well known mathematical objects such as sets and sequences" [Lamb88]. Logic programming theories to handle these objects are just now being developed. Logic programming languages that use those theories and handle these objects are not yet available. Therefore using logic programming to generate test data based on abstract models is not yet feasible.

5.4. Summary

In this chapter we have described and justified the assumptions inherent in our methodology. They are:

Software Testing Assumptions:

(1) We are testing an ADT.

(2) There is no coincidental incorrectness.

(3) The ADT is testable.

Abstract Data Type Assumptions:

(1) The language L(S) exists.

(2) The ADT contains only total functions.

(3) The abstract machine on which we run works correctly.

(4) The functions are primitive recursive.

Algebraic Specifications:

(1) The specifications exist.

(2) The specifications are of the form outlined in Section 3.2.1.

(3) The specifications are correct.
(4) The axioms of the specifications are consistent.
(5) The axiomatization is sufficiently complete.

For a completely automated system there would be one final requirement for our ADT specification. We would require a complete procedure for generating terms that satisfy all equations in the predicates of the axioms.

6. A NEW METHOD FOR TEST CASE GENERATION USING PROLOG

We have found PROLOG very useful for software test case generation. The general approach of our method is to use an algebraic specification of abstract data types to be tested and a series of tools based on logic programming to derive a series of functional test data. Previous researchers [Bouge85b, Bouge86, Wild86] have developed one method for generating test data from algebraic specifications. We found this method produced overly large data sets and could not handle, in finite time, a wide variety of predicates that may appear in algebraic specifications of ADTs. Examples of such problem predicates might be $\neq$, >=, < or is_empty. The only predicates handled by those previous systems are simple equality and boolean operations such as AND and OR.

In this chapter we present our method for using PROLOG to produce test cases for abstract data types. In Section 6.1 we outline what we call instance classes and sub-instance classes of ADTs, and why they are important for ADT testing. In Section 6.2 we present our general method for testing these instance classes and sub-instance classes. In Section 6.3 we present our method for using PROLOG to test ADTs. In Section 6.4 we give a brief example using our new method and show how it handles some major problems in a useful and consistent manner. We will show that our method:

(1) allows predicates other than equality to be tested;

(2) prevents infinite searching for equality/inequality predicates; and

(3) allows non-C-type TOI functions inside the traces being tested.

6.1. Instance Classes and Sub-Instance Classes

Weyuker and Ostrand [Weyuker80] significantly extended some preliminary work by Meyers [Meyers74, Meyers76] by introducing the concept of revealing sub-domains as a basis for white box software testing. They describe a subset of a program's input domain as revealing if the existence of one incorrectly processed input implies that all of that subset's elements are processed incorrectly. Their intent was to partition the program's domain "in such a way that all elements of an equivalence class are either processed correctly or incorrectly." While it is recognized that finding such a partition is in general as undecidable as finding a proof of correctness, Weyuker and Ostrand argued that "testing in terms of restricted subdomains allows us to concentrate on probable local errors, and increases the likelihood of finding good tests" [Weyuker80]. Our intent is also to partition the program's domain in such a way that all elements belong to exactly one sub-domain. We recall that our testing goal is different from the white box goal. Our goal is to test that the software system does "everything it is supposed to " rather than "only what it is supposed to." Thus rather than having "subsets such that the existence of one incorrectly

processed input implies that all of that subset's elements are processed incorrectly" as Weyuker and Ostrand did, we are interested *in subsets such that the existence of one correctly processed element implies that all of that subset's elements are processed correctly.*

Exhaustively testing every combination of input values for a software system is usually impossible. Therefore, to say something about the overall correctness of a program on the basis of some tests requires the user to extrapolate from a finite set of test cases to an infinite number of possible inputs. Commonly a user will say "The system passed these 200 test cases; therefore it will probably work properly all the time." The quality of the testing method used determines how convincingly a user can make that extrapolation. Any extrapolation from a finite set of information such as "the system passed these 200 tests" to a statement about something infinite such as "working properly all the time," requires making an "infinitary" [Bouge85a] hypothesis. In our case that extrapolation rests on the following hypothesis:

OUR INFINITARY HYPOTHESIS:

Given a language L, a set S, and an instance class I of L(S):

$[\forall I_i \text{ where } I_i \text{ is a sub-instance class of } I : \exists x\ x \in I_i \mid OK_b(x)] \rightarrow$
$[\forall I_i \text{ where } I_i \text{ is a sub-instance class of } I : \forall x\ x \in I_i \mid OK_b(x)]$

We can view this hypothesis as saying that if a programmer correctly implemented some elements in every possible sub-instance class of an instance class then he did

not incorrectly implement other instances in those sub-instance classes. The quality of test sets produced by our methodology can be expected to vary with the validity of that hypothesis.

Our infinitary hypothesis will allow us to more readily partition the elements of our TOI into subsets such that the existence of one correctly processed input in each set implies that all of the subset's elements are processed correctly. Now we group the elements of the TOI together such that:

(1) the series of TOI function calls that generate them is the same;

(2) the *observable functionality* that these elements should exhibit in a correct implementation of the ADT is the same.

By testing an implementation with one instance from a group of instances of the TOI, where that group has the characteristics given in (1) and (2), we show that the implementation gives the correct results for at least some elements of that subset. We can use our infinitary hypothesis to say that if the programmer implemented the correct functionality for some elements of the subset then he did not implement incorrect functionality for other elements of the subset. Thus all the elements of the subset function correctly ($\therefore \forall y \in X \ (OK(y))$). Therefore we may now make the uniformity hypothesis as presented in Figure 6.1.

Uniformity Hypothesis:

Given a subset D_i of a domain D:

$$(\exists(d)\in D_i \;\; OK(d)) \rightarrow (\forall d\in D_i \;\; OK(d))$$

This hypothesis holds for any subset with the group characteristics given in (1) and (2).

Algebraically:

i) $x_1 \in X$	Group X has properties (1) and (2) above
ii) $OK(x_1)$	Result of testing
iii) $(\exists x_1 \in X \mid OK(x_1)) \rightarrow \forall y \in X \;\; \overline{(\overline{OK}(y))}$	Infinitary Hypothesis
iv) $\exists x \in X \mid (OK(x))$	1 & 2
v) $\forall z \in X \;\; (\overline{OK}(z) \; OR \; OK(z))$	something is correct or it is not
vi) $\therefore \; \forall y \in X \;\; \overline{(\overline{OK}(y))}$	3 & 4
vii) $\therefore \; \forall y \in X \;\; (OK(y))$	5 & 6

Figure 6.1: Uniformity Justification

This uniformity hypothesis allows us to extend the results of testing one element of a subset of the TOI to the whole subset. We can make this extension as long as the subset has the group characteristics given in (1) and (2).

Grouping elements of the TOI into groups such that "the series of TOI function calls that generate them is the same" and then further partitioning these groups to form sub-groups such that "the observable functionality these elements should exhibit in a correct implementation is the same," is where we introduce "instance classes"

and "sub-instance classes" to our testing methodology. An instance class is made up of instances of the TOI whose traces contain the same series of TOI function calls. For example the elements of our List type[11] given in example 6.1 are members of the same instance class.

---

removeDups(AddElt(AddElt(initlist(),5),747))
removeDups(AddElt(AddElt(initlist(),99),99))
removeDups(AddElt(AddElt(initlist(),123),456))

Example 6.1

To represent an instance class we take the series of function calls for these instances and replace their non-TOI operands by variables. For example, the above elements are members of the instance class represented by :

remove Dups(AddElt(AddElt(initlist(),I1),I2))

Thus, instance classes define a grouping on the set of all possible ways to produce elements of the TOI. That grouping satisfies property (1) outlined earlier in this section.

In Section 2.3.4 we defined a sub-instance class as a group of elements of the TOI that are members of the same instance class and are specified to have the same

[11]The algebraic specification of our example ADT "List" is given in Appendix I.

observable functionality. That is, they contain all elements of an instance class with the same *expected* output functions. For example, the instances presented in example 6.1 are members of two sub-instance classes. The O-type functions for our List type, as specified in Appendix I, that produce "observable" functionality are: emptylist, getElt, size, and includes. Table 6.1 outlines the results of the O-type functions when applied to our three instances :

Trace	I1=5 ; I2=747	I1=99 ; I2=99	I1=123 ; I2=456
emptyList(removeDups(AddElt(AddElt(initlist(),I1),I2)))	false	false	false
getElt(removeDups(AddElt(AddElt(initlist(),I1),I2)))	I2	I2	I2
size(removeDups(AddElt(AddElt(initlist(),I1),I2)))	2	1	2
includes(removeDups(AddElt(AddElt(initlist(),I1),I2)),I1)	true	true	true
includes(removeDups(AddElt(AddElt(initlist(),I1),I2)),I2)	true	true	true
includes(removeDups(AddElt(AddElt(initlist(),I1),I2)),I3)	true	true	true

Table 6.1.

Note that the first and third instances produce the same observable functionality and that the second instance's observable functionality is different. Thus the first and third instances are members of the same sub-instance class and the second is a member of a different sub-instance class. If we group elements of the TOI into sub-instance classes then:

(1) every element of the TOI will appear in exactly one group;

(2) all elements of any group will be generated by the same series of TOI function calls;

(3) all elements of any group will exhibit the same *observable functionality* in a correct implementation of the ADT.

Now we may apply the argument shown in Figure 6.1. The result of that argument allows us to state that by partitioning the TOI into sub-instance classes, we have partitioned the program domain into "subsets such that the existence of one correctly processed input implies that all that subset's elements are processed properly."

6.2. Testing a Sub-Instance Class

In this section we describe how to test a given sub-instance class. Determining which sub-instance class to test next will be discussed in Chapter 8. For the purposes of this section we will assume we have been told which sub-instance class to test.

We recall that an instance class can be viewed as a series of TOI function calls with the non-TOI operands replaced by variables. From our discussion in the previous sections, it follows that given an instance class representation to test, we must determine what sub-instance classes exist for that instance class and then check that the observable functionality of an element of each of these sub-instance classes is consistent with the ADT's specification. From here on we will refer to "checking that the observable functionality of an element is consistent with the ADT's specification" as "testing" an element. Thus to test an instance class we must determine all its sub-instance classes and test an element from each of these sub-instance classes.

Checking the observable functionality of an element of a sub-instance class involves applying all applicable O-type functions to that element and checking that the result the implementation gives, R_i, is consistent with the expected result R_s. Algorithm 6.1 gives an outline for this process.

---

1) Given a sub-instance class trace γ
2) For each O-type function
    2.1) Determine R_s for each legal o-γ combination
        by following the procedure given in Section 6.3.
    2.2) Determine R_i for each legal o-γ combination
        by giving those function calls to the implementation.
    2.3) $R_s = R_i$?

Algorithm 6.1.

Determining an implementation result, R_i, is simply a matter of giving the appropriate set of TOI function calls and input values to the implementation and recording the output values. Thus, the keys to testing an instance class are :

(1) determine the sub-instance classes

(2) determine R_S for each sub-instance class.

PROLOG has proven particularly useful in accomplishing these two tasks.

6.3. An Updated Test Case Generation Method Using PROLOG

In this section we describe our method for using PROLOG to accomplish the two key testing tasks of determining the sub-instance classes of an instance class, and determining the expected result *R*, for each sub-instance class. We have found that previous PROLOG methodologies [Bouge85b, Choquet86, Wild86, Bouge86] could not be used in our general testing method because of a limitation we call "the constraint problem." In Section 6.3.1 we describe that problem. We next present our solution to that problem and, in Section 6.3.2, we will present how we have implemented our solution with PROLOG. In Section 6.3.3 we will outline an unexpected problem with using PROLOG in our more general testing approach. We will then present our solution to that problem, and show how we have implemented that solution with PROLOG. Finally in Section 6.3.4 we contrast our work with that of previous authors.

To begin with, we specify in Figure 6.2 a slightly more complex type, called Bag, with more observable functionality than our type Queue which was specified in Figure 2.1. Bag is more complex than the Queue because the "removeDups" operation depends on the values of non-TOI (*integer* in this case) inputs. Figure 6.3 gives the PROLOG specification for the ADT specified in Figure 6.2. The PROLOG specification was produced by applying the equivalences outlined in Section 3.2.3 to the algebraic specifications in Figure 6.2.

Type bag

SYNTAX

initBag()	->bag
addElt(bag,integer)	->bag
removeElt(bag,integer)	->bag
emptyBag(bag)	->boolean
sizeBag(bag)	->integer
includesElt(bag,integer)	->boolean
removeDups(bag)	->bag

SEMANTICS

Declare b1:bag,b2:bag,i1:integer,i2:integer

1) removeElt(initBag(),i1)	=initBag()
2) removeElt(addElt(b1,i1),i2)	=IF I1=I2 THEN b1 ELSE addElt(removeElt(b1,i2),i1)
3) emptyBag(initBag())	=true
4) emptyBag(addElt(b1,i1))	=false
5) sizeBag(initBag())	=0
6) sizeBag(addElt(b1,i1))	=sizeBag(b1)+1
7) includesElt(initBag(),i1)	=false
8) includesElt(addElt(b1,i1),i2)	=IF I1=I2 THEN true ELSE includesElt(b1,i2)
9) removeDups(initBag())	=initBag
10) removeDups(addElt(b1,i1))	=IF includesElt(b1,i1) THEN removeDups(b1) ELSE addElt(removeDups(b1),i1)

END.

Figure 6.2: Algebraic Specification of Type: BAG

```
removeElt(initBag,I1,initBag).
removeElt(addElt(B1,I1),I2,B1):- I1=I2 .
removeElt(addElt(B1,I1),I2,addElt(B2,I1)):- !,removeElt(B1,I2,B2).


emptyBag(initBag,true).
emptyBag(addElt(B1,I1),false).


sizeBag(initBag,0).
sizeBag(addElt(B1,I1),X):- !,sizeBag(B1,Y), X is Y+1.


includesElt(initBag,I1,false).
includesElt(addElt(B1,I1),I2,true):- I1=I2.
includesElt(addElt(B1,I1),I2,X):- !,includesElt(B1,I2,X).


removeDups(initBag,initBag).
removeDups(addElt(B1,I1),X):- includesElt(B1,I1,true), removeDups(B1,X).
removeDups(addElt(B1,I1),addElt(X,I1)):-  !,removeDups(B1,X).
```

Figure 6.3: PROLOG Specification of Type: BAG

The specification in Figure 6.3 is also a PROLOG program. If we load this program into a PROLOG interpreter we may give the interpreter goals of the form

$$oef\ (cf_1(cf_2(cf_3 \cdots)),\ Answer)$$

Where:

- cf_1, cf_2 and cf_3 are all C-type functions.
- *oef* is a non-C-type (O or E -type) TOI function
- *Answer* is an unbound PROLOG variable.

The interpreter will attempt to find a solution to this goal by using the clauses of the program as replacement rules to bind a value or variable to "Answer." Figure 6.4 shows several such queries and the output from a PROLOG interpreter.

Notes:

(1) Queries 1 and 2 give us the R_S values false and 1 for applying two of the O-type functions to the particular instance class addElt(initBag(),I1).

(2) Query 3 gives the two sub-instance classes corresponding to I1=I2 and I1≠I2 for the instance class removeElt(addElt(initBag(),I1),I2).

(3) Query 4 shows that there is only one sub-instance class associated with the instance class removeDups(addElt(initBag(),I1)).

```
yes

1)?- emptyBag(addElt(initBag,I1),Answer).

Answer = false,
I1 = _70 ? ;
no

2)?- sizeBag(addElt(initBag,I1),Answer).

Answer = 1,
I1 = _68 ? ;
no

3)?- removeElt(addElt(initBag,I1),I2,Answer).

Answer = initBag,
I1 = _72,
I2 = _72 ? ;

Answer = addElt(initBag,_72),
I1 = _72,
I2 = _92 ? ;
no

4)?- removeDups(addElt(initBag,I1),Answer).
Answer = addElt(initBag,_74),
I1 = _74 ? ;
no
```

Figure 6.4: PROLOG Queries for Type: BAG

6.3.1. The Constraint Problem

When we examine the specification in Figure 6.2, the only non-TOI predicate used is equality. It is used in axioms #2 and #8. These axioms translate, in part, to the second and ninth clauses in the PROLOG specification in Figure 6.3. These

clauses impose the constraint that I1=I2. This constraint gives rise to the first sub-instance class found in query #3 in Figure 6.4. This method of using PROLOG as described in previous papers [Bouge85b, Bouge86, Wild86] has been shown to work for several different sample ADT specifications. Unfortunately all those examples only allowed the equality predicate in the specification. Predicates such as ">" ">=" "≠" were never mentioned, much less demonstrated. It has been suggested [Choquet86] that these other predicates are unique and therefore will require special handling. We disagree with that suggestion.

In the specification in Figure 6.2 when we say "I1=I2", we mean the integer value of I1 must be the same as the integer value of I2. In the clauses of the PROLOG specification, the PROLOG interpreter takes "I1=I2" to mean I1 and I2 are to be unified. These two meanings are quite different. As a result of unifying I1 and I2 we do in fact ensure I1 and I2 have the same value, if they have one at all. Thus it is possible to get away with translating "I1=I2" in an algebraic specification into "I1=I2" in a PROLOG specification even though they do not mean the same thing. The important point is that equality is a unique predicate, and this uniqueness allows for a simple translation. This trick will not work for any other predicates. Thus, for a testing system that will work on more than a handful of examples, we have found that a general method for handling constraints that arise from any predicate is needed.

We note that many PROLOGs do have ">" and "\==" (not equal) symbols. The former fails if its operands are not arithmetic expressions, that is, it will not work for unbound variables. The later means "cannot currently be instantiated to ....". Neither of these are what we need.

6.3.2. A More General Constraint Handling Method.

In this section we outline a new, more general, method of handling general predicates and their associated constraints, for test case generation. We then present a new set of translation equivalences to implement this method. The results presented here focus on the problems of handling constraints in logic programming for test case generation. The reader is referred to [Jaffar87a, Jaffar87b] for discussions on the more general problems in constraint logic programming.

We begin by noting that in the equivalences in Section 3.2.3 we are already adding a variable (y in equivalence #1) to the axioms, to communicate the "answer" between sub-goals, and to return the "answer" to the parent clause. Thus it is reasonable to add other variables through the translation procedure to communicate information in addition to the final specified output result (R_S).

If we look at axioms such as #3 and #6 in the specification in Figure 6.2, we see that these axioms give information about R_S *for a particular class of traces*. Axioms such as #2 and #8 also give information about R_S but this information is subject to a

constraint: (I1=I2) or (not(I1=I2)). That is , these axioms also give information about the constraints C_{RS} that must be true for R_S to be valid. We propose to *explicitly* communicate this information between sub-goals, and to the parent clause, rather than *implicitly* communicate it, as has been done before.

We will introduce a variable called "constraints in " (CI) through the translation procedure. CI will contain the constraints the parent goal must currently satisfy. This variable will be used to communicate to each sub-goal those constraints the parent (and therefore the sub-goal itself) must currently satisfy. We also introduce through the translation procedure a variable called "constraints out" (CO). This variable will communicate back up to a parent goal those constraints this particular solution of the sub-goal must satisfy. Note that for any sub-goal the constraints coming in (CI) must be implied by the constraints going out (CO), i.e.,

$$CO \rightarrow CI$$

As pointed out above, axioms such as #2 and #8 in Figure 6.2 give rise to constraints of the form (I1=I2) and (not(I1=I2)). To allow a more general handling of predicates, so that we may uniformly handle predicates in addition to equality, we propose to view these constraints in the form:

I1 op1 I2

AND

I1 op2 I2

where op1 is "=" and where op2 is "≠"

We can now introduce a new set of equivalences that allow a consistent handling of all non-TOI predicates by a PROLOG interpreter. First we note that

$$f(x) = If \;\; C \;\; THEN \;\; A_1 \;\; ELSE \;\; A_2$$

can be viewed as

$$C \Rightarrow f(x) = A_1$$

$$\overline{C} \Rightarrow f(x) = A_2$$

Under a PROLOG formalism these axioms are written as :

$$f(x) = A_1 :- C(x)$$

$$f(x) = A_2 :- \overline{C}(x)$$

Our transformation into horn clauses is performed by using the following equivalences.

1) $f(g_1(x_1,\ldots,x_{n1}),g_2(\alpha_1,\ldots,\alpha_{n2}),\ldots,g_m(\gamma_1,\ldots,\gamma_{nm}),z_1,\ldots,z_p) = A \;:-\; C(X)\;. \;\Leftrightarrow$

$F(y_1,y_2,\ldots,y_n,z_1,\ldots,z_p,A,CI,CO)$:-

$G_1(x_1,\ldots,x_{n1},y_1,CI,Co_1)$,

$G_2(\alpha_1,\ldots,\alpha_{n2},y_2,Co_1,Co_2)$,

.

.

.

$G_m(\gamma_1,\ldots,\gamma_{nm},y_m,Co_{m-1},Co_m)$,

$add_constraint(C(X),Co_m,CO)$.

2) $f(g_1(x_1,\ldots,x_{n1}),g_2(\alpha_1,\ldots,\alpha_{n2}),\ldots,g_m(\gamma_1,\ldots,\gamma_{nm}),z_1,\ldots,z_p)=A$:- $\Leftrightarrow$

$F(y_1,y_2,\ldots,y_n,z_1,\ldots,z_p,A,CI,CO)$:-

$G_1(x_1,\ldots,x_{n1},y_1,CI,Co_1)$,

$G_2(\alpha_1,\ldots,\alpha_{n2},y_2,Co_1,Co_2)$,

.

.

.

$G_m(\gamma_1,\ldots,\gamma_{nm},y_m,Co_{m-1},CO)$.

"add_constraint(C,L1,L2)" simply adds the constraint C to the list of constraints L1 giving a new list of constraints L2.

An example using these equivalences is given in Section 6.4.

There are several advantages to our new method of translating axioms to horn clauses.

(1) This method does not require the PROLOG interpreter to implement any of the predicates that are used in the axioms. Thus the range of predicates that the tester may use is no longer limited by the PROLOG interpreter.

(2) The interpretation of the meaning of predicates such as ">" and "≠" are left to the user. By using this new procedure the PROLOG interpreter directly returns the constraints instead of simply implying them. If we look at query #3 in Figure 6.4, we see that $I1 \neq I2$ is implied in the second answer. That constraint is not given directly.

(3) We will no longer get infinite depth-first recursion for predicates such as $X \neq Y$. With our method the interpreter no longer tries to "solve" $X \neq Y$ by enumerating all possible X's, all possible Y's, and then returning pairs where $X \neq Y$. The interpreter simply treats the constraint $X \neq Y$ as a fact.

(4) This method makes the use of integers in our specification a lot easier: we do not have to view each integer as a set of "base" and "successor" functions as has been done previously [Bouge85b, Choquet86].

(5) Finally, abstract data types tend to be hierarchical in nature. That is, they are built on top of each other. Thus ADT's will be built and specified using types more complicated than just integers. This method allows for the use of arbitrarily complicated predicates from these more complicated types to be used in the specification of new higher types.

6.3.3. The Problem of More General Traces

As outlined in Sections 6.1 and 6.2, our testing method calls for generating sub-instance classes and expected results for traces made up of any of the TOI operations. This is significantly different from previous users of PROLOG for testing. We showed at the beginning of Section 6.3 that previously, goals had to be of the form:

$$oef\,(cf_1(cf_2(cf_3 \cdots)),\ Answer)$$

where: cf_1, cf_2, and cf_3 can ONLY be C-type functions.

Thus previous methods only allowed non-C type functions to appear as the outermost function of the trace. For example, the goals in Figure 6.5 simply return "no" for the PROLOG specification in Figure 6.3. This problem arises because the axioms of the initial algebraic specification define the non-C-type functions of the ADT in terms of the C-type functions, but the C-type functions are only implicitly defined. There are no explicit axioms for C-type functions.

```
yes

5)?- removeElt(removeElt(addElt(addElt(initBag,I1),I2),I3),Answer).
no

6)?- removeElt(removeElt(addElt(addElt(initBag,I1),I2),I3),I4,Answer).
no

7)?- removeDups(removeElt(addElt(addElt(initBag,I1),I2),I3),Answer).
no

1?-
```

Figure 6.5

To be unable to test traces of arbitrary orders of C-type and non-C-type functions is a significant limitation of an ADT testing method. We have had to develop new translation equivalences that differentiate between C-type and non-C-type functions so that our testing method can handle traces that may contain arbitrary orders of functions. This was not required before, as previous researchers did not require that their methodologies handle general traces.

Figure 6.6 contains our final new series of translation equivalences. They are an extension of those outlined in Sub-section 6.3.2. This extension allows for arbitrary traces in PROLOG by recognizing that C-type and non-C-type functions are specified differently in the algebraic specifications, and that both types should be allowed in an arbitrary trace.

---

1) $f(g_1(x_1,\ldots,x_{n1}),g_2(\alpha_1,\ldots,\alpha_{n2}),\ldots,g_m(\gamma_1,\ldots,\gamma_{nm}),z_1,\ldots,z_p)=A$:- $C(X)$. $\Leftrightarrow$

$F(y_1,y_2,\ldots,y_n,z_1,\ldots,z_p,A,CI,CO)$:-

$G_1(x_1,\ldots,x_{n1},y_1,CI,Co_1)$,

$G_2(\alpha_1,\ldots,\alpha_{n2},y_2,Co_1,Co_2)$,

•

•

$G_m(\gamma_1,\ldots,\gamma_{nm},y_m,Co_{m-1},Co_m)$,

add_constraint$(C(X),Co_m,CO)$.

-where F & G_x are relation symbols corresponding to f & g_x,
and CI,CO,Co_x and y_x are PROLOG variables.
-For any axioms with constraints " $C(X)$".
-For any non-C-type functions f,$g_1,\ldots,g_m$.

2) $f(g_1(x_1,\ldots,x_{n1}),g_2(\alpha_1,\ldots,\alpha_{n2}),\ldots,g_m(\gamma_1,\ldots,\gamma_{nm}),z_1,\ldots,z_p)=A$:- $\Leftrightarrow$

$F(y_1,y_2,\ldots,y_n,z_1,\ldots,z_p,A,CI,CO)$:-

$G_1(x_1,\ldots,x_{n1},y_1,CI,Co_1)$,

$G_2(\alpha_1,\ldots,\alpha_{n2},y_2,Co_1,Co_2)$,

.

.

$G_m(\gamma_1,\ldots,\gamma_{nm},y_m,Co_{m-1},CO)$.

-where F & G_x are relation symbols corresponding to f & g_x, and CI, CO, Co_x and y_x are PROLOG variables.
-For any axioms without constraints.
-For any non-C-type functions $f, g_1,\ldots,g_m$.

3) $F(h_1(x_1,\ldots,x_{n1}),h_2(\alpha_1,\ldots,\alpha_{n2}),\ldots,h_m(\gamma_1,\ldots,\gamma_{nm}),z_1,\ldots,z_p)\cdots$:- $\cdots\Leftrightarrow$

$F(y_1,y_2,\ldots,y_n,z_1,\ldots,z_p)\cdots$:-

y_1 is $H_1(x_1,\ldots,x_{n1})$,

y_2 is $H_2(\alpha_1,\ldots,\alpha_{n2})$,

.

.

y_m is $H_m(\gamma_1,\ldots,\gamma_{nm})$, $\cdots$

-where F & H_x are relation symbols, H_x corresponding to h_x, and y_x are PROLOG variables.
-For any non-C-type functions f.
-For any C-type function h_x.

4) $\cdots$:- $\cdots F(g(x_1,\ldots,x_n),z_2,\ldots,z_m,A,CI,CO)\cdots\Leftrightarrow$

$\cdots$:- $\cdots G(x_1,\ldots,x_n,y,CI,C_a)$, $F(y,z_2,\ldots,z_m,A,C_a,CO)$

-where F corresponds to any TOI function (C-type or non-C-type). and G is a relation symbol corresponding to g.
-For any non-C-type function g, at any inner level of nesting on the right hand side of the PROLOG implication.

Figure 6.6: A New Set of Translation Equivalences

Notes:

I) Equivalence 3) handles C-type functions in a way consistent with the handling of non-C-type functions in equivalences 1) and 2). Equivalence 3) is not necessary if nesting of non-C-type functions inside $h_1 \cdots h_m$ does not happen or is not allowed.

II) There is no equivalent to 4) for C-type functions.

III) Equivalence 3) need not be applied if the results of the non-C-type functions inside $h_1 \cdots h_m$ do not affect the axiom.

IV) "add_constraint" is simply a PROLOG predicate that adds a constraint to an already existing set of constraints in a consistent manner; this set of constraints could be empty.

6.3.4. Comparison to Previous Work

This new method of using PROLOG to generate test cases allows three significant improvements in the solution of the larger problem of testing the implementations of abstract data types. Those improvements are:

(1) we can test *any* combination of TOI functions;

(2) our method does not repeatedly generate the same test case over and over;

(3) we can test against all the axioms of an ADT at the same time.

Previous test case generation methodologies were limited in the types of functions that could occur in test cases. They were *"limited to those terms of size less than k that are combinations of constructors. The number of generated instantiations is then considerably decreased."* [Bouge85b] While we agree that decreasing the size of a test set is desirable if nothing else is sacrificed, but that is not the case here. As we showed in Section 4.5.1, previous researchers needed to make a hidden assumption equivalent to "All E-type functions and their combinations work correctly" to produce a justifiable test set. They needed to make that assumption because their method was limited to terms that are combinations of C-type functions only. Our new translation equivalences remove the requirement of looking at only "combinations of constructors." We feel that a testing method that requires us to assume that a potentially large part of a software system works correctly without ever testing it, defeats the point of doing testing. The fact that our PROLOG method will handle terms made up of any valid combination of ADT functions means our overall ADT testing method does not have those limitations.

Another significant point about our PROLOG method is that it does not repeatedly generate the same test case over and over. In previous methodologies, given the constraint a≠b, where a and b were specified to be of a particular type, a PROLOG interpreter would give all the a x b combinations where a ≠ b. That is (1,2), (1,3), (1,4), (1,5), (1,6) ..... Our method properly returns two sub-instance classes (a,b | a=b)

and (a,b | a≠b).

We have eliminated potentially infinite backtracking for all inequalities (< and ≠ are examples of inequalities) by treating the constraints they imply as conditions rather than problems to be solved. Previous methods required a PROLOG interpreter to determine all possible solutions for an inequality. There maybe an infinite number of solutions to these inequalities and thus this may lead to infinite backtracking. Our method returns the inequality as a condition that is either true or false. Therefore a PROLOG interpreter will only have a finite number of solutions to check via backtracking.

Finally, at a higher level, we can view the work of previous researchers as studying *"how to test an implementation of a data type against a property (an axiom) which is required by the specification"* [Bouge86]. With our new methodology we can now view all the properties (axioms) of the specification as a unit. This is better than having to look at each individual property (axiom) separately.

6.4. An Extended Example

To demonstrate our method, we extend our previous Bag example to a new type called Bag-n with a function whose axioms could not previously be handled. That function will be called "negs" and will return true if the bag contains a negative integer and returns false otherwise. We will call the new type "Bag-n." The additional axioms for this new function are given in Figure 6.7. Figure 6.8 gives the PROLOG specification of our Bag-n type. Note that :

(1) constraints are now explicitly handled;

(2) we no longer use "=" for equality; we use "eq", and that this is consistent with the other predicates (ge,lt,ne);

SYNTAX

negs(bag)	->boolean

SEMANTICS

11) negs(initBag())	=false
12) negs(addElt(b1,i1))	=IF i1<0 THEN true ELSE negs(b1)

Figure 6.7: Additional Axioms for Type Bag-n

---

(3) there is no problem adding the "negs" function.

Figure 6.9 is a PROLOG listing for several goals given to the interpreter after the program in Figure 6.8 had been loaded. Note that for all three instance classes the interpreter returns the correct result (Ans) for each {trace x constraint} pair as well as the constraints for each of the sub-instance classes.

The first query in Figure 6.9 shows that our method correctly handles the new function "negs" which contained a predicate other than simple equality (in this case "less than"). There are two sub-instances: one where the integer is less than zero, and the other occurs when that integer is greater than or equal to zero.

The second query shows that our method correctly handles traces that have inner functions other than C-type functions. These traces simply could not be handled before. In this case the trace is:

size(removeDups(addElt(addElt(initBag,I1),I2)))

Note that "removeDups" is an E-type function.

The third query is a combination that has inner non-C-type functions and the new "negs" function that could not be run with previous methods. Note that for this trace there will be three test cases, one when the two integers are equal and two different cases where they are not equal.

```
removeElt(initBag,I1,initBag,CI,CI).
removeElt(addElt(B1,I1),I2,B1,CI,CO):- add_constraint(I1 eq I2,CI,CO).
removeElt(addElt(B1,I1),I2,addElt(B2,I1),CI,CO):- !,removeElt(B1,I2,B2,CI,Ca),
                add_constraint(I1 ne I2,Ca,CO).


emptyBag(initBag,true,CI,CI).
emptyBag(addElt(B1,I1),false,CI,CI).



sizeBag(initBag,0,CI,CI).
sizeBag(addElt(B1,I1),X,CI,CO):- !,sizeBag(B1,Y,CI,CO), X is Y+1.



includesElt(initBag,I1,false,CI,CI).
includesElt(addElt(B1,I1),I2,true,CI,CO):- add_constraint(I1 eq I2,CI,CO).
includesElt(addElt(B1,I1),I2,X,CI,CO):- !,includesElt(B1,I2,X,CI,Ca),
                add_constraint(I1 ne I2,Ca,CO).


removeDups(initBag,initBag,CI,CI).
removeDups(addElt(B1,I1),X,CI,CO):- includesElt(B1,I1,true,CI,Ca),
                removeDups(B1,X,Ca,CO).
removeDups(addElt(B1,I1),addElt(X,I1),CI,CO):-  !, includesElt(B1,I1,false,CI,Ca),
                removeDups(B1,X,Ca,CO).


negs(initBag,false,CI,CI).
negs(addElt(B1,I1),true,CI,CO):- add_constraint(I1 lt 0,CI,CO).
negs(addElt(B1,I1),Ans,CI,CO):- negs(B1,Ans,CI,Ca), add_constraint(I1 ge 0,Ca,CO).
```

Figure 6.8: PROLOG Specification: BAG-n

```
yes

| ?- negs(addElt(initBag,I1),Ans,[],COnstraints).

Ans = true,
COnstraints = [_62 lt 0],
I1 = _62 ? ;

Ans = false,
COnstraints = [_62 ge 0],
I1 = _62 ? ;

no

| ?- removeDups(addElt(addElt(initBag,I1),I2),Y,[],Ca), sizeBag(Y,Ans,Ca,COnstraints).

Ans = 1,
COnstraints = [_92 eq _112],
Ca = [_92 eq _112],
I1 = _92,
I2 = _112,
Y = addElt(initBag,_92) ? ;

Ans = 2,
COnstraints = [_92 ne _112],
Ca = [_92 ne _112],
I1 = _92,
I2 = _112,
Y = addElt(addElt(initBag,_92),_112) ? ;

no

| ?- removeElt(addElt(initBag,I1),I2,Y,Ca,COnstraints),negs(Y,Ans,[],Ca).

Ans = false,
COnstraints = [_72 eq _92],
Ca = [],
I1 = _72,
I2 = _92,
Y = initBag ? ;

Ans = true,
COnstraints = [_72 ne _92,_72 lt 0],
```

```
Ca = [_72 lt 0],
I1 = _72,
I2 = _92,
Y = addElt(initBag,_72) ? ;

Ans = false,
COnstraints = [_72 ne _92,_72 ge 0],
Ca = [_72 ge 0],
I1 = _72,
I2 = _92,
Y = addElt(initBag,_72) ? ;

no
```

Figure 6.9 Sample PROLOG Run for Bag-n

6.5. Conclusion

In this chapter we have outlined a new method for using PROLOG to generate test data for abstract data types. This method allows two major advances.

(1) We can now test instances whose traces contain any combination of TOI functions. Previously published methods only allowed testing of traces whose inner functions are C-type functions. We have shown that this amounts to assuming all E-type functions and their combinations are correct. To require a software tester to assume that a potentially large part of the software to be tested is correct is a significant limitation. By developing a method that can handle any combination of TOI functions, we have removed that limitation.

(2) We can now handle, in finite time, specifications that contain predicates other than equality. As we showed in this chapter, previous methods could handle the equality predicate because of its unique properties. Our method, by treating

constraints as facts, prevents infinite searching, and allows the use of any predicate in the ADT specification.

Our example showed that this method correctly handles predicates other than equality and traces that contain functions other than C-type functions as well as C-type functions. This significantly expands the class of ADTs that we can test in this way as it allows us to test ADTs that have E-type functions.

7. A COMPUTATIONAL MODEL FOR ABSTRACT DATA TYPES

In this chapter we outline a new model of the computations performed by abstract data types. This model was developed to give an understanding of what it means to "test" an element of an ADT. We begin by showing how sub-instance classes of the TOI can be viewed as *trace* × *constraint* pairs. This view will be useful for describing and understanding our model.

The specified observable functionality of an element of the TOI is dependent on:

(1) the function calls in the trace which generated that element; and

(2) the values of the non-TOI variables that were used as input to those functions.

An instance class may have several sub-instance classes if the observable functionality of its elements is dependent on characteristics of the non-TOI input elements. For example:

> If we have an operation $f_a(x)$ in our abstract data type that is defined as follows:
>
> $f_a(x) = f_w(x)$ *IF* $x > 0$
> $f_a(x) = f_y(x)$ *IF* $x = 0$
> $f_a(x) = f_z(x)$ *IF* $x < 0$
>
> Then the instance class whose trace is $f_a(x)$ will have three associated sub-instance classes.

Therefore in our computational model can view a sub-instance class, whose elements are all specified to have the same functionality, as a *trace* × *constraint* pair. For example, in our "Bag" type the instance class whose uninstantiated trace is:

removeDups(addElt(addElt(initBag(),i1),i2))

would have two sub-instance classes defined by the following *trace* × *constraint* pairs:

$$removeDups(addElt(addElt(initBag(), i1), i2)) \quad \times \quad i1=i2$$

$$removeDups(addElt(addElt(initBag(), i1), i2)) \quad \times \quad i1\neq i2$$

7.1. A Canonical Form for ADTs

When we presented the theoretical foundation for our specification directed software testing methodology we stated that we will be working with a sufficiently complete set of axioms. We showed that any sufficiently complete axiomatization may be viewed as a set of replacement or re-write rules. If an axiomatization is sufficiently complete for a language L(T), which is the language defined by the abstract type T=[{V},{S+O}], then every term in the language must be assigned a meaning by the axiomatization. We can view these axioms as re-write rules; every trace of ADT functions can be assigned a canonical form by those re-write rules. The axioms of the algebraic specification of an ADT define the results of O-type and E-type functions in terms of C-type and N-type functions. Thus a *sufficiently complete*

algebraic specification of an ADT can be used to re-write any valid *fully instantiated* trace of ADT functions as an equivalent trace containing only C-type and N-type ADT functions.

If we take a "re-written" form of a trace and replace all non-TOI values by variables then we get what we call the canonical form of that fully instantiated trace.

Definition (Canonical Form):

> Let L be a language, S be a set and {variables} be a set of variables. An instance i_c of the language L({S $\cup$ {variables}}) whose symbols are $p_1 p_2 p_3 \ldots p_n$ is a canonical form if $\forall$ p [(p $\in$ C) $\vee$ (p $\in$ N) $\vee$ (p $\in$ {variables})], where C is the set of C-type functions in L, and N is the set of N-type functions in L.

For example, with our "List" type (see Appendix I):

(1)

removeDups (*addElt* (*addElt* (*initList* (),747),747))

via axiom 23 rewrites to:

removeDups (*addElt* (*initList* (),747))

which via axiom 23 reduces to:

addElt (*removeDups* (*initList* ()),747)

which via axiom 22 reduces to:

addElt (*initList* (),747).

Replacing non-TOI values with variables we get:

addElt (*initList* (),*J* 1).

Thus *addElt* (*initList* (),*J* 1) is the canonical form for *removeDups* (*addElt* (*addElt* (*initList* (),747),747))

(2)

removeDups (*addElt* (*addElt* (*initList* (),123),456))

via axiom 23 rewrites to:

addElt (*removeDups* (*addElt* (*initList* (),123)),456)

which via axiom 23 rewrites to:

addElt (*addElt* (*removeDups* (*initList* ()),123),456)

which via axiom 22 rewrites to:

addElt (*addElt* (*initList* (),123),456).

Replacing non-TOI values with variables we get:

addElt (*addElt* (*initList* (),*J* 1),*J* 2).

Thus *addElt* (*addElt* (*initList* (),*J* 1),*J* 2) is the canonical form for *removeDups* (*addElt* (*addElt* (*initList* (),123),456))

7.2. Equivalence Spaces

We now look at (trace, constraint) pairs. For any *fully instantiated* trace that produces an element of the TOI, we can build an equivalent canonical *trace* containing only C-type functions according to the set of specification *constraints* that it satisfies. Table 7.1 lists three fully instantiated traces, their instance class, their canonical form, and the constraints that the input variables had to satisfy according to the specifications. It should be noted:

(1) that the first and second traces are part of the same instance class but do not reduce to the same canonical form; and

(2) that the second and third traces have very different operation sentences but reduce to the same canonical form.

We introduce the new concept for ADTs of equivalence space as a space that contains all the (trace, constraint) pairs that can reduce to the same canonical form. An equivalence space is the set of all sub-instance classes that have the same functionality specified for them.

We can view an equivalence space as containing all traces whose canonical representation is the same series of ADT-function calls. We note that some traces cannot be reduced, that is, they are already in a canonical form. We view

Trace	Instance Class	Canonical Form	Constraints
removeDups (*addElt* (*addElt* (*initBag* ()*,*6)*,*6))	*removeDups* (*addElt* (*addElt* (*initBag* ()*,i* 1)*,i* 2))	*addElt* (*initBag* ()*,i* 1)	*i* 1=*i* 2
removeDups (*addElt* (*addElt* (*initBag* ()*,*5)*,*6))	*removeDups* (*addElt* (*addElt* (*initBag* ()*,i* 1)*,i* 2))	*addElt* (*addElt* (*initBag* ()*,i* 1)*,i* 2)	*i* 1≠*i* 2
removeElt (*addElt* (*addElt* (*addElt* (*removeElt* (*initBag* ())*,*5)*,*6)*,*7))	*removeElt* (*addElt* (*addElt* (*addElt* (*removeElt* (*initBag* ())*,i* 1)*,i* 2)*,i* 3))	*addElt* (*addElt* (*initBag* ()*,i* 1)*,i* 2)	NONE

Table 7.1

these as prototypical elements of the equivalence space. We call these structures *"equivalence* spaces" because their members are *specified* to be equivalent, and, in a *correct* implementation of the ADT would be equivalent. For example, in our "bag" type example:

removeDups (*addElt* (*removeElt* (*addElt* (*addElt* (*initBag* (),747),99),99),747))

AND

addElt (*initBag* (),747)

are specified to have the same functionality, and in a *correct* implementation would be equivalent to each other. They both should produce bags of one element and that element should be 747. Thus they are in the same *equivalence* space.

Definition (Equivalence Space):

> Let L be a first order language, S a set, and A an $L'(S)$-theory where $L \subset L'$, and $E \subset L(S)$. If $(\forall i \in E(\forall j \in L(S)((i =_A j) \rightarrow (j \in E))))$ then E is an equivalence space for L(S).

where $i =_A j$ means i and j reduce to the same canonical form according to theory A.

Thus there is one equivalence space associated with each canonical form of an ADT, and under the sufficiently complete assumption (Section 5.3.4), every fully instantiated trace of ADT functions that produces elements of the TOI falls into at least one of these equivalence spaces. Our specification consistency requirement (Section 5.3.3) ensures that they fall into at most one of these equivalence spaces.

7.3. Computation Space

We define the computation space for a particular ADT as the space that contains all the equivalence spaces for that ADT, and the transitions between these equivalence spaces. These transitions result when we apply a valid TOI-function to a member of an equivalence space. Since that means we are applying a valid ADT function to a trace of valid ADT functions, we get a new trace of ADT functions. As we stated, the result of this trace is an element of the TOI. Since any valid trace of the ADT that returns an element of the TOI is a member of exactly one equivalence space, this new trace is an element of some equivalence space in that same computation space. Thus the application of C-type and E-type functions of an ADT may lead to transitions between equivalence spaces in the computation space of that ADT. For an ADT whose set of equivalence spaces is $\{S_E\}$ and whose C-type and E-type functions are $\{F_C\}$ and $\{F_E\}$, the computation space S_C is a relation:

$$S_c \subseteq \{\{S_E\} \times \{\{F_C\} \cup \{F_E\}\}\}.$$

Figures 7.1, 7.2, 7.3 and 7.4 show portions of the computation space for our "Bag" example. They show the equivalence spaces that may be reached by traces of length 2,3,4, and 5 or less, respectively, for our Bag-type example. The ovals in these figures represent equivalence spaces, and are labeled with their canonical form. The arcs represent TOI-function applications. In Figure 7.1 we can see that

traces of length 2 can reach two possible equivalence spaces: removeDups(initBag()) and removeElt(initBag()) reach the lower equivalence space; addElt(initBag(),i1) reaches the upper equivalence space. Note that the equivalence space reached is independent of any input values. Therefore all of these traces represent instance classes with only one sub-instance class.

In Figure 7.2 we can see that traces of length 3 or less can reach one of three possible equivalence spaces. Again, note that the equivalence space which is reached by a trace of length 1, 2, or 3 is solely dependent on that trace's *operation sentence* (see Chapter 8 or Glossary) with no constraints on the non-TOI variables, and is independent of any externally supplied data.

In Figure 7.3 we can see the equivalence spaces that traces of length 4 or less can reach. Note that the trace

removeDups(addElt(addElt(initBag(),i1),i2))

can reach one of two possible equivalence spaces depending on which set of constraints ($i1=i2$ *OR* $i1\neq i2$) the variables i1 and i2 satisfy.

In Figure 7.4 can see how traces of longer length can reach several different equivalence spaces depending on constraints the non-TOI input values satisfy. In this figure that means constraints for which values of i1, i2, i3 are equal. For example:

removeDups(addElt(removeDups(addElt(addElt(initBag(),i1),i2)),i3))

may take any one of five sets of arcs depending upon which set of constraints variables i1, i2 and i3 satisfy.

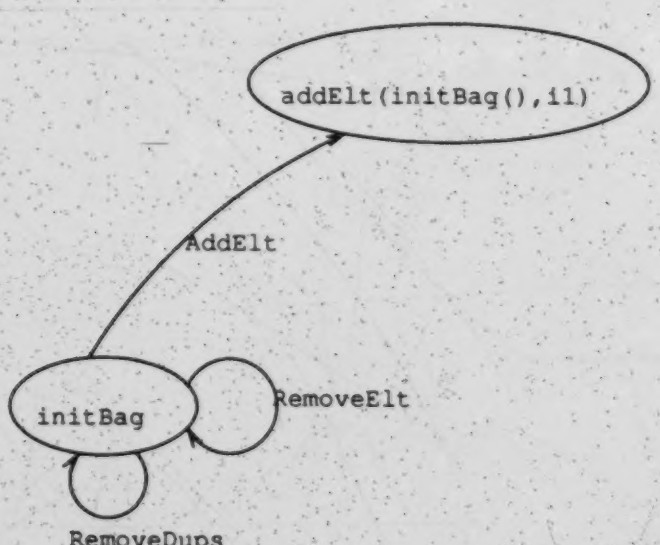


Figure 7.1: Traces of Length 2 or Less for Bag Type

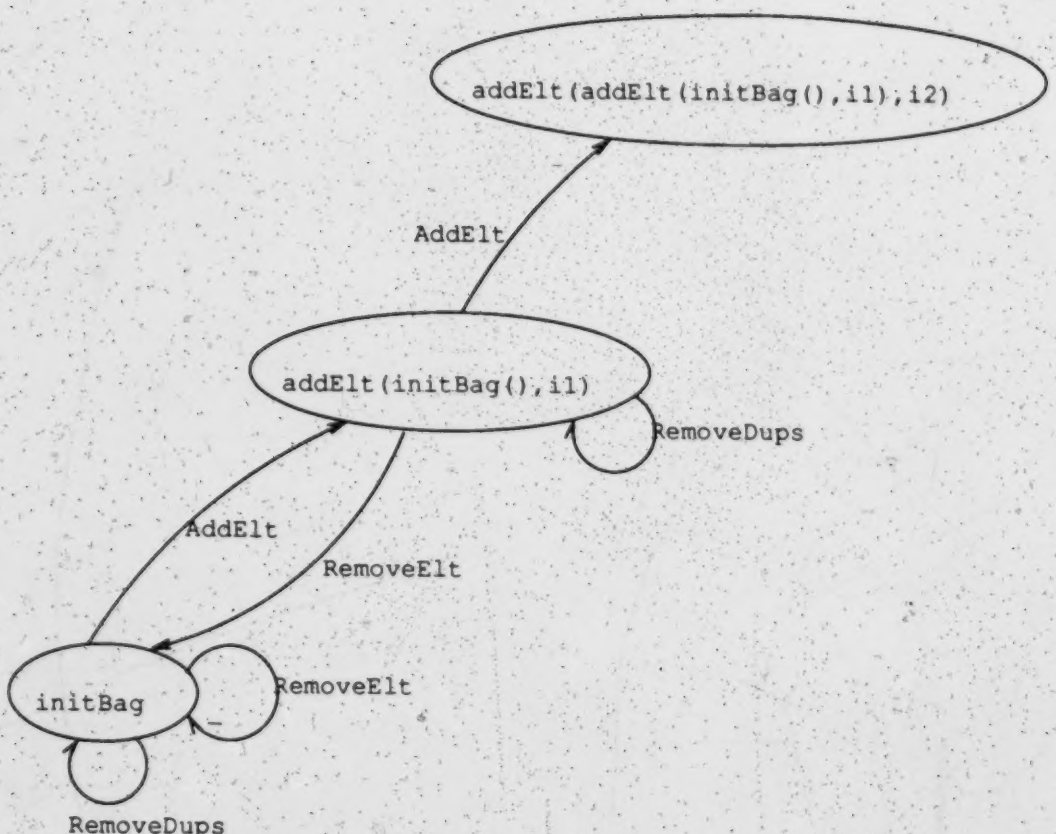


Figure 7.2: Traces of Length 3 or Less for Bag Type

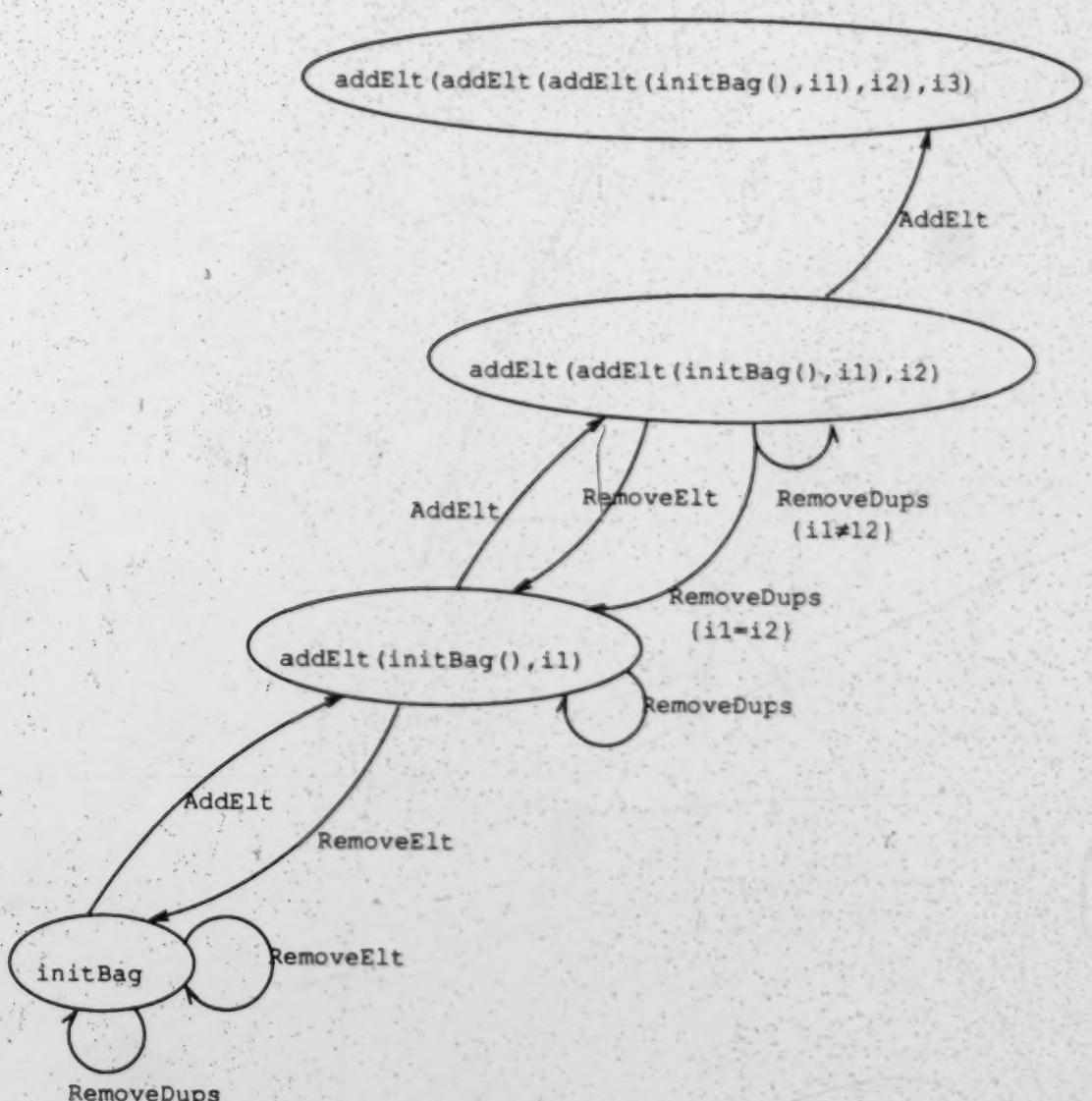


Figure 7.3: Traces of Length 4 or Less for Bag Type

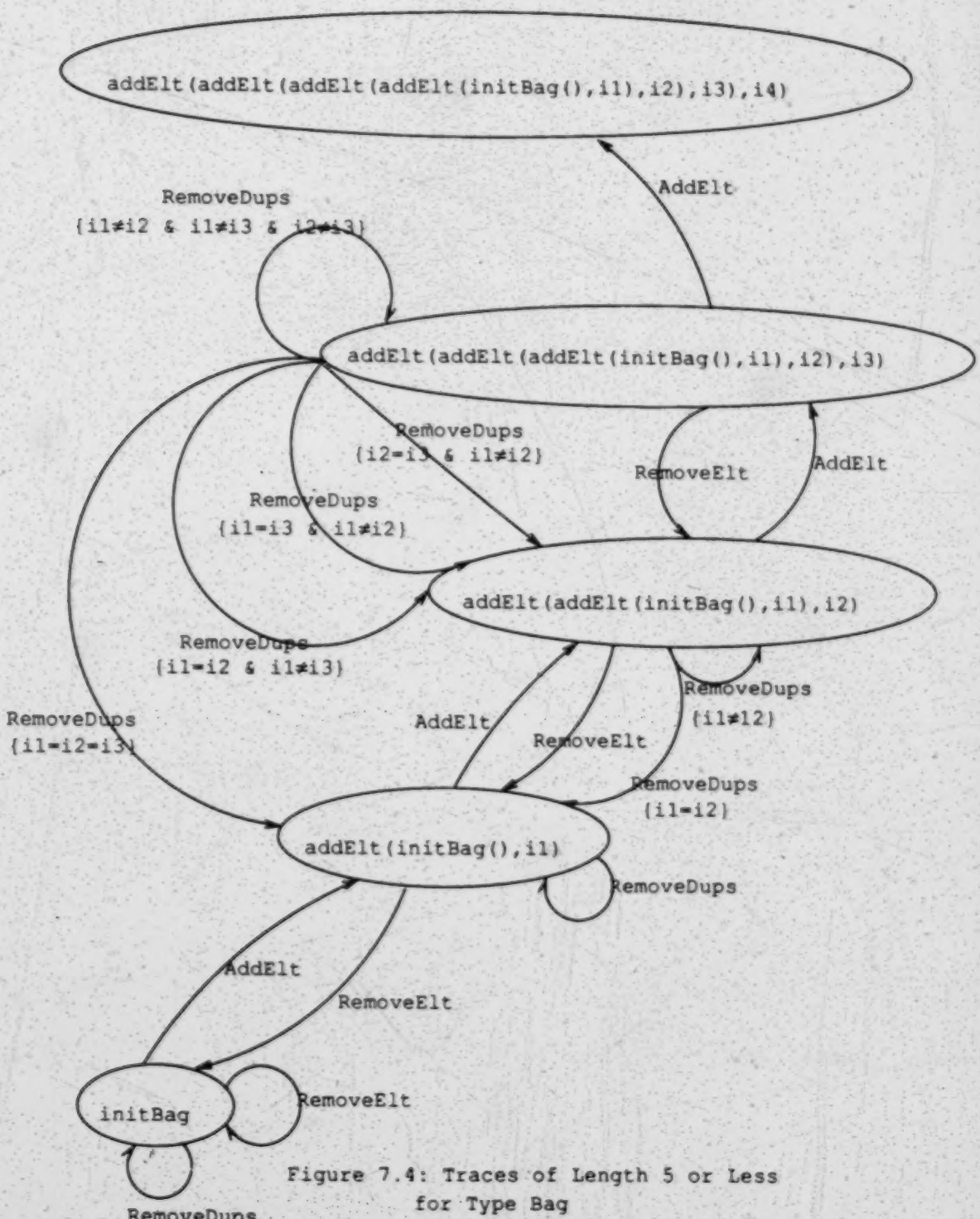


Figure 7.4: Traces of Length 5 or Less
for Type Bag

7.4. Implications for ADT Testing

Our ADT testing method involves producing a series of instance classes $\Gamma=<\gamma_1,\gamma_2,\cdots>$ for the ADT being investigated, and then determining all sets of constraints on the non-TOI values in the operations of each γ_i that will cause functionally different behavior. That is, for each instance class γ_i, we must determine the functionally different sub-instance classes. For each of these sub-instance classes we select an element and check that the observable functionality for that element is consistent with that ADT's specification. This is done by applying all possible O-type functions to that element.

In terms of our computation space model, our testing method selects an element, and that element necessarily belongs in *one* equivalence space. We then apply all the mappings that map from the equivalence space of that element to an output value. Since output values are not members of an equivalence space, we are applying all mappings that take us out of the computation space for that particular ADT.

Thus when we say we have "tested" an element of the TOI we are saying that we have checked:

(1) that it has been mapped correctly into its equivalence space; and

(2) that the observable functionality (output values) of *that* equivalence space are correct.

This new model of the computations of ADTs and its commensurate insight into the testing of elements of the ADT may be used as a basis for ordering instance classes to produce test sets. This will be outlined in Chapter 8.

8. TEST SET GENERATION

In this chapter we describe how to generate an *ordered* battery of tests. We will call an ordered battery of tests a test set. These test sets are made up of a series of instance classes to be tested. We will often be concerned with the sequence of ADT function calls in the trace of an instance or instance class.

Definition (Operation Sentence):

An operation sentence for an ADT is:

Base Case: a valid application of a single operation for that ADT;
or
Recursive Case: an application of a valid operation of that ADT to an operation sentence of that ADT.

Example:

Assume an ADT with three operations V= { $f_a(a_1, \cdots, a_m)$, $f_b(b_1, \cdots, b_n)$, $f_c(c_1, \cdots, c_p)$}. For clarity we will denote the operation sentence $f_a(a_1, \cdots, a_m)$ as f_a and the operation sentence $f_a(a_1, \cdots a_{x-1}, f_b(b_1, \cdots, b_n), a_{x+1}, \cdots, a_m)$ as $f_a f_b$.

Then for this ADT the set of operation sentences is that subset of the set $\{f_a, f_b, f_c, f_a f_a, f_a f_b, f_a f_c, f_b f_a, f_b f_b, \ldots\ldots, f_a f_b f_c f_a f_c, \ldots\ldots\}$ which represents allowable sequences of operation applications.

8.1. Goals of the Testing Method

The goal of our software testing method is to detect the presence of failures in a software system (see chapter 4). A failure is an observable event where the

performance of a system is inconsistent with its specifications. A failure should not be confused with an error, which is a piece of information which when processed by the system may produce a failure.

It is important to realize that to attempt to make a software system fail we are going to apply a sequence of tests; one test is to be applied after the other. We cannot guarantee that as more tests are run more failures will be detected; but it is still desirable to generate tests such that the more ADT operations we execute, the more likely we are to detect failures. Note we are not saying "the more tests we successfully run, the more likely the software is correct." As we showed in chapters 2 and 4, the later approach is not practical because testing cannot show the absence of errors, only their presence. Note also that we are producing a test case series whose *elements are not independent of each other*. The $n+1^{st}$ choice of which instance class to test next will depend on which n instance classes have already been tested.

8.2. A General Method For Testing Abstract Data Types

As we have previously stated, our testing method involves producing a series of instance classes $\Gamma=<\gamma_1,\gamma_2,\cdots>$ for the ADT being investigated, and then determining what constraints on the non-TOI values in the operations of each γ_i produce functionally different behavior. That is, for each instance class we determine the different sub-instance classes. We have shown that these differences in

functionality (for example: includesElt(B,i)= true, or, includesElt(B,i)= false, depending on the value of "i") are delineated by constraints defined in the axioms of the specification. For each of these sub-instance classes, we select an element and check that the *observable functionality* (as defined in Section 2.2) for that instance is correct, that is, we check that the output from the O-type functions of the implementation when applied to that instance are consistent with that ADT's specification.

8.2.1. Instance Classes

A significant advance in our testing method, and indeed one of the keys to our approach, is the ordering of the instance classes that are to be tested for a given ADT. We construct the series $\Gamma=<\gamma_1,\gamma_2,\gamma_3,\cdots>$ for each $n=1,2,3,\cdots$ by determining which instance class with associated trace γ_n has the largest ratio of possible failures to testing cost, *given that* $\gamma_1 \ldots \gamma_{n-1}$ *have executed in a manner consistent with the ADT's specification.*

8.2.1.1. Possible Failure Set (PFS)

Given that the goal of our testing methodology is to make the software which we are testing fail, and that the tester has only limited resources to achieve that goal, we would like to test the system as economically as possible. To "economically" test we need some concept of cost and benefit for testing. To model

how an ADT can fail we introduce in this sub-section the concept of a Possible Failure Set or PFS for an ADT. We will show that there are two basic ways an ADT can fail, and we define the PFS to include those two ways. We will use this failure model to develop a measure of the benefit we get from running a given test.

As we discussed in Chapter 2, for an abstract data type it is possible to identify a set of constructor functions or C-functions. The idea is that any trace that contains E-type functions can be equivalently written using only C-type functions. In Chapter 7 we defined a canonical representation for ADTs and developed the concept of an equivalence space. If we replace all instantiated non-TOI elements with variables in a canonical representation we define an "equivalence space," and can view it as containing all traces that, according to the specifications, can be reduced to the same series of ADT-function calls. An example of this might be "all stacks of depth 4."

There are several important characteristics of these equivalence spaces. First, we can view the ADT-functions as mappings between equivalence spaces. For example, adding an element to a list of length 3 to produce a list of length 4 is a mapping from a particular 3-dimensional equivalence space to a particular 4-dimensional equivalence space. Similarly, sorting a list of length 3 is a mapping from a 3-dimensional equivalence space into a 3-dimensional equivalence space.

The second characteristic of an equivalence space is that there are generally an infinite number of traces that may lead to it. We can view these traces as routes to that equivalence space within the computation space of that ADT. For example a queue of length three may be generated by adding three elements to an initial empty queue, by adding four elements to an initial empty queue and deleting one, or by adding five elements to an initial empty queue, deleting three elements, and then adding another element.

The third characteristic of equivalence spaces is that together, they contain every element of the TOI for an ADT. For example, every queue produced by our ADT Queue as defined in Figure 2.1 is a member of an equivalence space of queues. In Section 7.2 we showed that every trace of ADT functions that produces elements of the TOI falls into at least one equivalence space and at most one equivalence space.

Guttag [Guttag80] has argued that any useful computation done by an implementation of an ADT gives a result that is not a member of the TOI, and that any ADT that does not return a non-TOI value is of no use. For testing purposes we extend this and recognize that any useful computation involving an ADT may be viewed as having two parts:

(1) creating a particular element of the ADT; and

(2) applying an O-type function to that element.

We have shown that every element of the TOI is a member of exactly one equivalence space. By definition, the *only observable* events for an ADT are the results of applying output functions to an element of the TOI. Since a failure is an *observable* event where the system violates its specification, the only way a failure can occur is by applying an output function to an element of the TOI as that is the only way to produce an observable event with an ADT. Therefore, for an ADT failure to occur we must have at least one of the following:

(1) an incorrect TOI element;

(2) an incorrect output function applied to a TOI element.

The only way an element of the TOI can be created is with a trace of ADT functions that leads to an equivalence space. In terms of our computation model, an output function is a mapping from an equivalence space to a non-TOI value. Therefore, from the two possible conditions for an ADT failure we find that the conditions for failure in our computation model are:

Computation Space Model of Conditions for an ADT Failure:

there must exist at least one equivalence space such that either
(1) one of the traces that lead to it contains a fault; or
(2) one of the mappings from it (to an element outside the computation space) contains a fault.

As stated previously, to test an element of the TOI we check its "observable behavior," that is, we apply all valid O-type (non-TOI) functions. These non-TOI mappings from an equivalence space are implicitly included when we test a route to an equivalence space. Thus we define the Possible Failure Set (PFS) for an ADT as the union of

(1) the set of different traces to each equivalence space, and

(2) the set of O-type function mappings from each equivalence space.

Definition (PFS):

$$PFS = \{\{S_E\} \times \{\{O\} \cup \{T\}\}\}$$

Where: $\{S_E\}$ is the set of all equivalence spaces
$\{O\}$ is the set of all O-type functions
$\{T\}$ is the set of all TOI traces.

Viewed in another way, the PFS contains the union of all the routes from the initial states of the ADT (for example an empty queue) into an equivalence space with all the ways out of an equivalence space, over the complete computation space of the ADT.

As we have already discussed for software testing in general, a fault is a defect which may generate an error, and an error is an item of information. From our definition of the PFS we can see that our testing method allows for faults in any arbitrary sequence of functions that produce an element of the TOI (any element of {T}) or any output function (any element of {O}). If we have a TOI trace whose operation

sentence is $T_1 = f_a f_b f_c \ldots . f_n$ then it is a member of {T} in our definition of the PFS. The trace $T_2 = f_b f_a f_b f_c \ldots . f_n$, which can be viewed as $T_2 = f_b(T_1)$ is also a member of {T}. Since T_1 and T_2 are separate members of {T}, we are treating the testing of T_2 as a separate event from the testing of T_1. Therefore we are not assuming that an error from the evaluation of T_1 will *necessarily* be propagated to the evaluation of $T_2 = f_b(T_1)$.

8.2.1.2. Testing an Instance class

The PFS is the set of all ways an ADT implementation can fail. Once we have tested an element of the PFS we know that the system will not fail for that element. Therefore, by successfully testing an element, we have removed it from the set of possible failures. The goal of our testing method may now be viewed as "removing elements from the possible failure set (PFS) as cheaply as possible." To be able to do this we need to be able to calculate two things:

(1) How many elements will be removed from the PFS by the testing of a particular instance class?

(2) How much will it "cost" to test that instance class?

In this section we describe a method we have developed for testing instance classes that uses the results of our investigation of testing sub-instance classes described in Section 6.2. With that method established, we investigate the "costs"

and "benefits" of testing an instance class in the following two sections.

From our definition of sub-instance class in Chapter 2 we know that a sub-instance class partitions an instance class such that every element of an instance class falls into exactly one of the sub-instance classes for that instance class. Therefore to test an instance class it is sufficient to test all its sub-instance classes.

To test an instance class we must test all its sub-instance classes. To test a sub-instance class we use our result from Chapter 6 (Figure 6.1) which used our infinitary hypothesis and states that "if the test is successful for one TOI element in a sub-instance class the program behaves correctly for any TOI element in this sub-instance class." That is, elements of our TOI generated by the same series of operations and whose non-TOI operands satisfy the same set of constraints behave in the same manner. Thus, if one element in a sub-instance class behaves correctly then we can expect all elements in that sub-instance class to not exhibit any failures. For testing this means that *to test a sub-instance class we select an element from that sub-instance class and check that its functionality is consistent with the ADT's specification.*

Since an ADT is specified in terms of its operations, the only functionality an element of the TOI exhibits is the non-TOI outputs from the O-type functions applied to that element. Thus, to check the functionality of an element of a sub-instance class we apply all O-type operations to that element's trace. We then

check that the output from those operations are consistent with the specifications of the ADT. Figure 8.1 outlines how to test an instance class. Note that Algorithm 6.1 is contained inside step 2) in Figure 8.1.

Given a trace T_γ of an instance class γ:

1) Determine all sub-instance classes of that instance class (i.e., determine the different sets of constraints)
2) FOR EACH SUB-INSTANCE CLASS:
    2.1) Generate an element satisfying all constraints for that sub-instance classes.
    2.2) FOR EACH O-TYPE OPERATION OF THE ADT:
        2.2.1) Execute the operations in that elements trace with the appropriate input values and apply the O-type operation to the result.
        2.2.2) Check that the non-TOI outputs are consistent with the ADT's specification
        2.2.3) IF they are consistent, continue; ELSE stop and report a failure.

Figure 8.1 : Algorithm to Test an Instance Class

8.2.2. Calculating The Relative Cost of Testing an Instance Class

In Section 8.2.1.2 we showed that if we test all the sub-instance classes of an instance class, then we have tested that instance class. Therefore the cost of testing an instance class is the cost of testing all its sub-instance classes. This in turn is the sum of the costs of applying all O-type operations to elements of that sub-instance class and the cost of creating those elements. In Chapter 5 we showed how we can not make any assumptions about how the ADT under consideration is implemented as that to some degree defeats the purpose of using ADTs. Thus, if an ADT has two TOI-operations, f_a and f_b, we do not assume that $f_a(a_1, \cdots a_n)$ is more or less expensive an operation to execute than $f_b(b_1, \cdots b_n)$. We can, however, assume that $f_b(b_1, \cdots f_a(a_1, \cdots, a_n), \cdots b_m)$ will be more expensive than $f_a(a_1, \cdots a_n)$. Thus we use the number of ADT function calls to test that instance class as our cost metric. Given an instance class whose associated trace operation sentence is $\gamma = f_a f_{a-} f_{a-2} \cdots f_1$, and a list $O=(o_1, o_2, \ldots, o_s)$ containing all applicable output functions for the ADT, the cost of testing that instance class is shown in Figure 8.2.

If we view the cost equation in Figure 8.2 at a higher level it makes intuitive sense. The 'n' term gives us the cost of one experiment (as defined in Chapter 2) for that instance class. The other term gives us the number of experiments that must be run to test that instance class. Therefore, their product gives us the cost of all experiments required to test that instance class.

$$n \times \sum_{i=1}^{s} (\#\ of\ (trace, constraint)\text{–}pairs\ \text{for}\ o_i \cdot \gamma)$$

# of operation calls for any element of the class

total # of sub-instance classes for that instance class

Figure 8.2 : The Cost of Testing an Instance Class γ

As will be shown later in Table 8.4, the number of test cases generated by applying an output function to a trace $(o_i \cdot \gamma)$ may vary depending on both the trace and the output function.

For the "List" type specified in Appendix I we may want to test:

removeDups(addElt(addElt(initList(),I1),I2)).

This particular example has also been used in Chapter 7. There are four O-type functions for this ADT. Table 8.1 shows the eight sub-instance classes that need to be tested for this instance class. As we can see, there are five ADT operation calls for each test. Those five are the O-type function call, one "removeDups" call, two "addElt" calls, and one "initList" call. Therefore the relative cost of testing this

o_i,γ	Constraints	Result
emptyList(γ)	None	False
getElt(γ)	I1≠I2	I2
getElt(γ)	I1=I2	I1
size(γ)	I1=I2	1
size(γ)	I1≠I2	2
includes(γ,I3)	I1=I3	True
includes(γ,I3)	I2=I3	True
includes(γ,I3)	I1≠I3 & I2≠I3	False

NOTE: γ = removeDups(addElt(addElt(initList(),I1),I2))

Table 8.1: Sub-instance Classes for o_i,γ

instance class is 5x8=40.

8.2.3. Calculating the Relative Benefit of Testing an Instance Class

Since our goal is to make the software system fail with a limited set of resources, we would like to exercise, and therefore remove from the PFS, as many elements as possible, yet hold the costs of testing to a minimum. As we showed in Section 8.2.1.2 the goal of our testing method may now be viewed as removing elements from the PFS as cheaply as possible. Thus the "benefit" of testing an instance class is the removal of some elements from the PFS. Therefore the metric we use in comparing the relative benefit of testing two different instance classes is the number of elements the testing of each removes from the possible failure set that have not previously been removed.

Given our "no coincidental incorrectness" assumption from Section 5.2, we can now identify three ways the testing of an instance class may affect the PFS. The first effect of testing an instance class whose equivalent operation sentence is $F = f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$ is the removal from the PFS of the trace $f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$ as a route to each of the equivalence spaces associated with each of the canonical forms of $f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$. To see this, recall that in Section 8.2 we showed that a trace may lead to more than one canonical form depending on the set of constraints satisfied by the input variables, and that every canonical form has an associated equivalence space. In Section 8.2.1.1 we defined the PFS for an ADT as PFS=$\{\{S_E\} \times \{\{O\} \cup \{T\}\}\}$. Now, for a tested trace "T" that has canonical forms C_{T1}, C_{T2}, C_{T3},... with associated equivalence spaces S_{E1}, S_{E2}, S_{E3},... we have removed (tested) $S_{E1} \times T$, $S_{E2} \times T$, $S_{E3} \times T$,... from the possible failure set. If we use the list example given in Appendix III and used in Chapter 7 we can see this. In Section 7.1 we showed that

removeDups(addElt(addElt(initList(),I1),I2))

can reduce to:

addElt(initList(),I1) AND

addElt(addElt(initList(),I1),I2)

Note that each of these reduced canonical forms has an associated equivalence space. In testing that instance class, we have removed one path to each of these equivalence

spaces from the PFS. Figure 8.3 is the portion of the computation space shown in Figure 7.4 with the two elements we have removed emboldened.

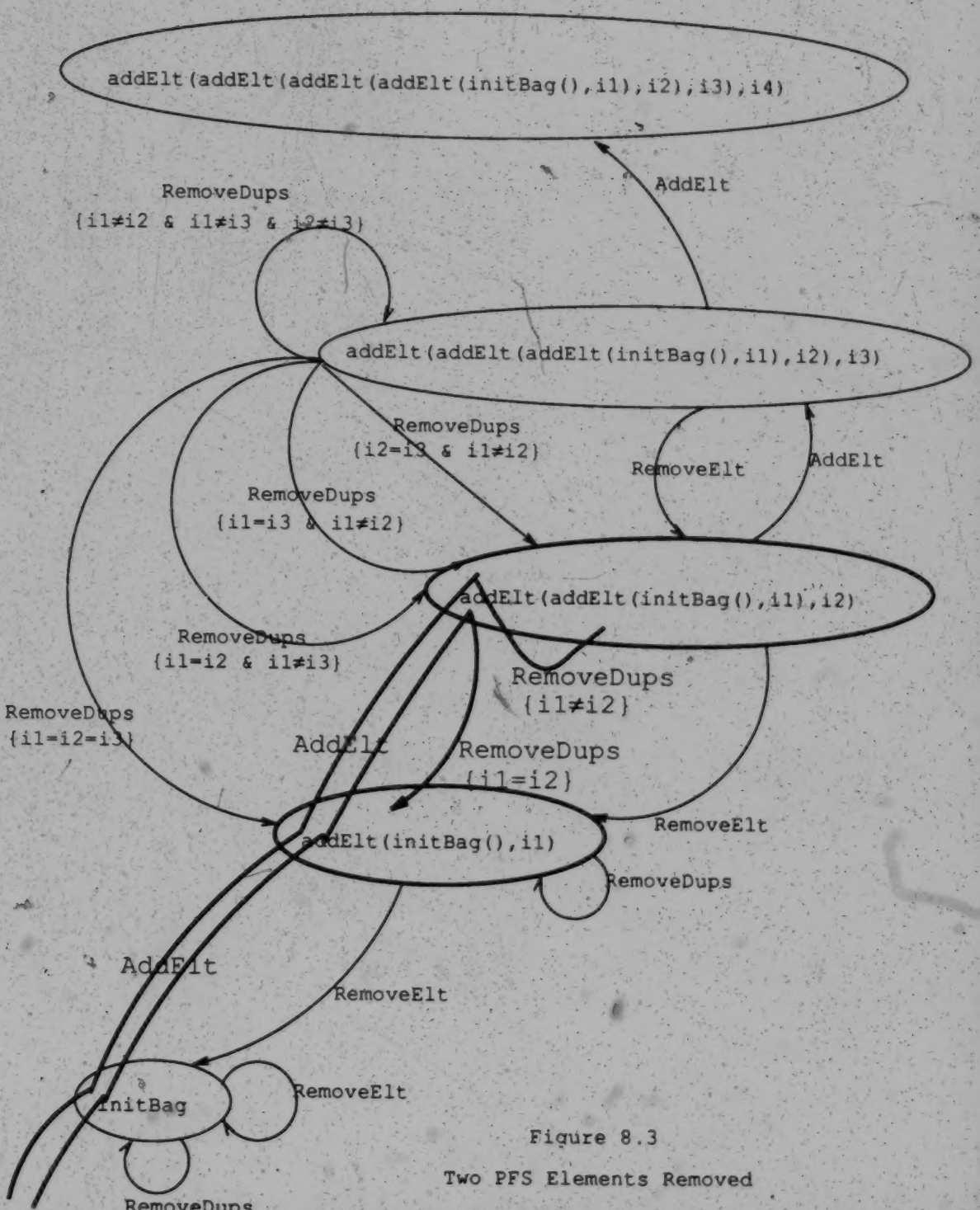


Figure 8.3

Two PFS Elements Removed

The second effect of testing an instance class is the removal of the mappings $o_i(E_f)$ from the PFS, where the o_i's are the O-type functions and the E_f's represent the equivalence spaces associated with each of the canonical forms of $f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$. To see this effect, recall that every instance "i" of an ADT is a member of exactly one equivalence space ($S_{Ei} \in \{S_E\}$). To test that instance involves applying all possible O-type functions $\{o_1, o_2, o_3, \ldots\}$ to that instance. Thus we have removed $S_{Ei} \times o_1, S_{Ei} \times o_2, S_{Ei} \times o_3, \ldots$ from $\{\{S_E\} \times \{\{O\} \cup \{T\}\}\}$. In terms of Figure 8.3, we have removed from the PFS the arcs from the two emboldened equivalence spaces and that lead to elements outside the computation space of the ADT. For this "list" example we have four O-type functions (emptyList, getElt, size, includes), thus 2x4=8 $S_E \times o_i$ elements are removed from the PFS.

The third effect the testing of an instance class has on the PFS is more subtle. It arises from the fact that if we have executed a trace, for example:

addQ (*addQ* (*deleteQ* (*addQ* (*initalQ* 0,*i* 1)),*i* 2),*i* 3)

we have also executed the inner traces

addQ (*deleteQ* (*addQ* (*initalQ* 0,*i* 1)),*i* 2) and *deleteQ* (*addQ* (*initalQ* 0,*i* 1)) , etc.

In such a case we may be able to say we have tested these inner traces. We can say this only if all the sub-instance classes of the inner trace have been preserved in the outer trace. This does not always happen. For example, in testing a trace such as

deleteQ (*deleteQ* (*addQ* (*addQ* (*initialQ* 0,*i* 1),*i* 2)))

we are essentially testing an empty queue which will exhibit very little functionality. In this case we cannot say we have also tested the inner trace

$$addQ(addQ(initialQ(),i1),i2)$$

as there would be sub-instance classes in that queue of two elements that would be lost upon deleting elements and thus would not be preserved in the larger trace.

For our *removeDups(addElt(addElt(initList(),I1),I2))* example in Figure 8.3 we want to know if we can remove any of the elements of the PFS associated with

(1) addElt(addElt(initList(),I1),I2);

(2) addElt(initList(),I1);

(3) initList().

The paths associated with each of these "inner" traces are labeled and emboldened in Figure 8.4. In this example all the constraints for testing *initList()* are included in the sets of constraints for testing *addElt(initList(),I1)*. All the sets of constraints for testing *addElt(initList(),I1)* are included in the sets of constraints for testing *addElt(addElt(initList(),I1),I2)*. Finally, all the sets of constraints for testing *addElt(addElt(initList(),I1),I2)* are included in the sets of constraints for testing *removeDups(addElt(addElt(initList(),I1),I2))*. Thus for this example we have three more $S_E \times T$ elements (one for each of the traces enumerated above) that are removed from the PFS.

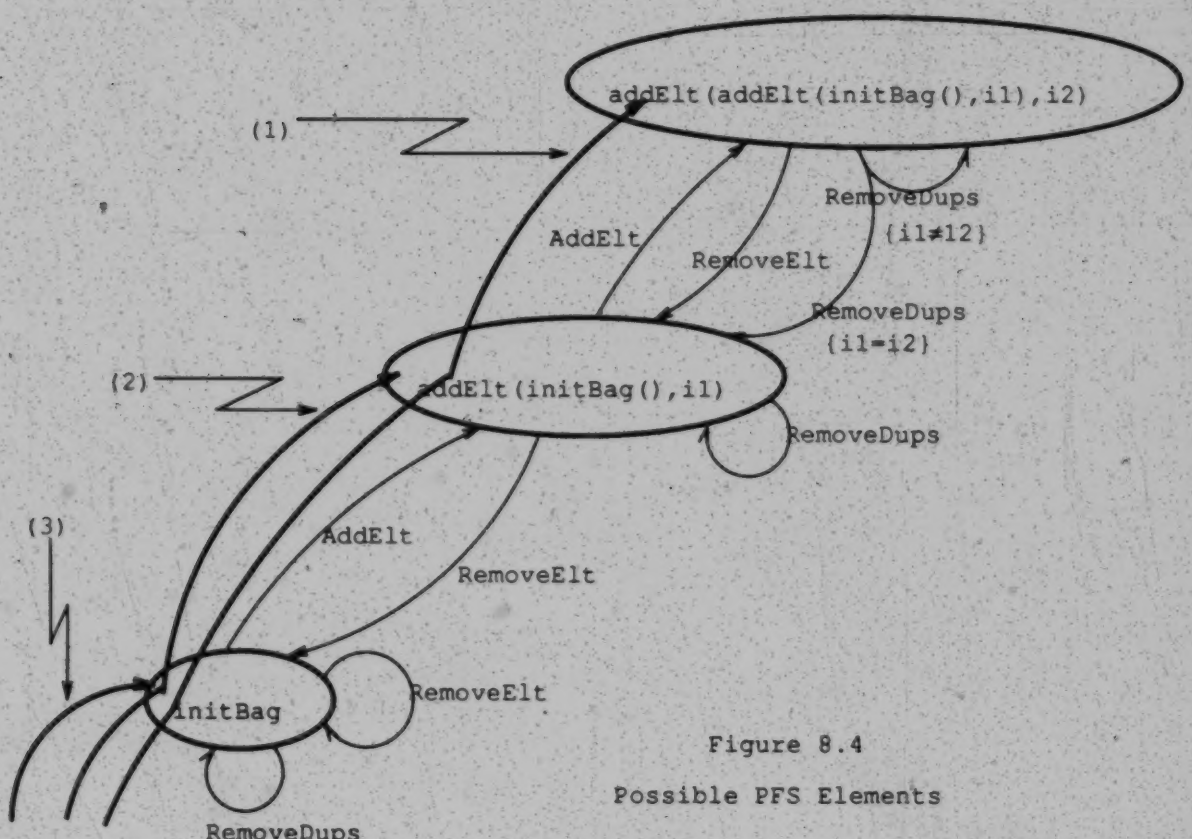


Figure 8.4

Possible PFS Elements

To characterize this third effect of testing in terms of our PFS model, we say that for testing an instance class whose equivalent operation sentence is $F = f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$, we may also remove the elements of the PFS that testing $f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$, would remove, if all the functional boundaries of all the domains of the sub-instance classes of $f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$, are preserved in $f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$.

An alternate view of this is: if in satisfying each individual set of constraints that determine the sub-instance classes in $f_n f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$, we have also satisfied all the individual sets of constraints in $f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$, then we have also tested $f_{n-1} f_{n-2} f_{n-3}, \ldots, f_1$.

An example of where we cannot say we have tested the "inner" traces of a particular element is:

deleteElt(addElt(initList(),I1)).

In this case all the constraints for testing addElt(initList(),I1) are not included in the constraints for testing deleteElt(addElt(initList(),I1)).

To summarize, there are three ways the testing of an instance class may affect the PFS:

(1) The paths associated with the trace of that instance class are removed form the PFS.

(2) The output functions that have been applied to the tested instances are removed from the PFS.

(3) Some of the smaller traces included in the trace of that instance class *may* also have been tested, and could therefore be removed from the PFS.

For our example instance class

removeDups(addElt(addElt(initList(),I1),I2))

we remove 2+8+3=13 elements form the PFS (assuming they have not already been removed by previous tests):

2 - from the first effect;
8 - from the second effect;
3 - from the third effect;

Thus the relative benefit of testing this instance class is 13.

8.2.4. An Algorithm For Testing ADTs

Our new method for testing an ADT can be described in general terms as repeatedly selecting and testing the untested instance class that is the most cost effective to test. Figure 8.5 gives our algorithm for testing an ADT.

Note that this algorithm is an instantiation of the general testing method for which we developed a theoretical foundation in Chapter 4. In that chapter we showed that at the highest level our methodology would "iteratively test instance classes of the ADT in the order defined by our complexity metric." Step 2 of our algorithm calculates its values as described in Sections 8.2.2 and 8.2.3. Note also that steps 3 and 4 incorporate the algorithm given in Figure 8.1, which in turn incorporates Algorithm 6.1 given in Section 6.2.

8.3. Example

Figure 8.6 gives an algebraic specification of an abstract data type called *"Bag-c"* (of integers). This *Bag-c* type is more complex than the Bag type we specified in Chapter 2. The N-type function is *initBag()*, which produces a new, empty, object of type *Bag-c*. The C-type function is *addElt(bag,integer)*. There are two E-type functions, the first is *removeElt(bag,integer)* which removes *one* element from the bag if that integer is in the bag, otherwise nothing is removed. The other E-type function is *removeDups(bag)*, which returns a bag of equal or smaller size than

1) Start with PFS containing all mappings and canonical forms.
2) (Re)-calculate which untested instance class has the highest "test value" Y. That is, the sentence F:

$$F = f_n f_{n-1} f_{n-2} \cdots f_1 (n > 0)$$

with the highest value Y:

$$Y = \frac{\#\ elements\ of\ PFS\ which\ are\ removed\ by\ testing\ F}{n \times \sum_{i=1}^{s} (\#\ of\ (trace, constraint)\text{-}pairs\ \text{for}\ o_i \cdot F)}$$

WHERE $O = (o_1, o_2, \ldots, o_s)$ is the set of all O-type functions.

3) Determine all sub-instance classes of that instance class (i.e., determine the different sets of constraints)
4) For each sub-instance class:
    4.1) Generate an element satisfying that instance classes constraints.
    4.2) For each O-type operation of the ADT:
        4.2.1) Execute the operations in that elements trace with the appropriate input values and apply the O-type operation to the result.
        4.2.2) Check that the non-TOI outputs are consistent with the ADT's specification
        4.2.3) IF they are consistent, continue ELSE stop and report a failure.
5) Remove appropriate members from PFS.
6) Go to step 2.

Figure 8.5 : Algorithm for testing an ADT

the given one. The returned bag contains the same integers as the given one except that any particular integer element appears only once.

Figure 8.7 gives the PROLOG program that we derived for our Bag-c type specification in Figure 8.6. For this example we built the tree of traces, to be evaluated for test cases, to a depth of six (i.e., we looked at all traces of length six or less).

The PROLOG program in Figure 8.7 was produced by applying the equivalences in Figure 6.6 to the algebraic specifications of our Bag-c type. That program was used to determine the number of test cases that resulted from separate sub-instance classes in each trace. From the number of test cases we determined the cost of testing each instance class using the method outlined in Section 8.2.2. The PROLOG program was also used to determine the benefit of testing each instance class as per Section 8.2.3. A detailed description of these PFS benefit calculations is given in Appendix IV. The algorithm in Figure 8.5 was then applied to produce the series of instance classes to be tested, and thus a series of test cases. Table 8.2 shows the instance classes that are tested by the first 30 test cases. Each time the instance class with the best benefit/cost ratio was selected. Once an instance class was selected, the appropriate elements of the PFS were marked as tested (step 5 in Figure 8.5). Then the benefit and benefit/cost values were recalcu-

Type bag-c

SYNTAX

initBag()	->bag
addElt(bag,integer)	->bag
removeElt(bag,integer)	->bag
emptyBag(bag)	->boolean
sizeBag(bag)	->integer
includesElt(bag,integer)	->boolean
removeDups(bag)	->bag

SEMANTICS

Declare b1:bag,b2:bag,i1:integer,i2:integer

1) removeElt(initBag(),i1)	=initBag()
2) removeElt(addElt(b1,i1),i2)	=IF i1=i2 THEN b1 ELSE addElt(removeElt(b1,i2),i1)
3) emptyBag(initBag())	=true
4) emptyBag(addElt(b1,i1))	=false
5) sizeBag(initBag())	=0
6) sizeBag(addElt(b1,i1))	=sizeBag(b1)+1
7) includesElt(initBag(),i1)	=false
8) includesElt(addElt(b1,i1),i2)	=IF i1=i2 THEN true ELSE includesElt(b1,i2)
9) removeDups(initBag())	=initBag
10) removeDups(addElt(b1,i1))	=IF includesElt(b1,i1) THEN removeDups(b1) ELSE addElt(removeDups(b1),i1)

END.

Figure 8.6: An Algebraic Specification for a Type Bag-c

```
removeElt(initBag,I1,initBag,CI,CI).
removeElt(addElt(B1,I1),I2,B1,CI,CO):- add_constraint(I1 eq I2,CI,CO) .
removeElt(addElt(B1,I1),I2,addElt(B2,I1),CI,CO):- removeElt(B1,I2,B2,CI,CX), add_constraint(I1

emptyBag(initBag,true,CI,CI).
emptyBag(addElt(B1,I1),false,CI,CI).

sizeBag(initBag,0,CI,CI).
sizeBag(addElt(B1,I1),X,CI,CO):- sizeBag(B1,Y,CI,CO), X is Y+1.

includesElt(initBag,I1,false,CI,CI).
includesElt(addElt(B1,I1),I2,true,CI,CO):- add_constraint(I1 eq I2,CI,CO).
includesElt(addElt(B1,I1),I2,X):- includesElt(B1,I2,X,CI,CX) add_constraint(I1 ne I2,CX,CO).

removeDups(initBag,initBag,CI,CI).
removeDups(addElt(B1,I1),X,CI,CO):- includesElt(B1,I1,true,CI,CX), removeDups(B1,X,CX,CO).
removeDups(addElt(B1,I1),addElt(X,I1),CI,CO):-  removeDups(B1,X,CI,CO).
```

Figure 8.7: A PROLOG Specification for a Type Bag-c

First 30 Tests for Search to Depth 6					
Trace Ordering	Trace Sentence	Level	# of Test Cases	PFS Elements Removed	B/C
1	E,D,I	3	3	5	5/9=0.56
2	A,D,I	3	4	4	4/12=0.33
3	E,E,D,D,E,I	6	3	5	5/18=0.28
4	D,E,D,E,E,I	6	3	4	4/18=0.22
5	E,D,E,D,D,I	6	3	4	4/18=0.22
6	D,D,D,E,A,I	6	7	9	9/42=0.21
7	A,A,A,A,I	5	7	6	6/35=0.17
Total			30	37	

NOTE: E= removeElt
D= removeDups
I= initBag
A= addElt

Table 8.2: Order of Traces Tested for Type Bag-c

lated for each trace and the next trace with the largest benefit/cost value was selected (step 2 in Figure 8.5). It is not necessary to repeatedly re-calculate the cost of testing each remaining instance class as that value does not change. The benefit for testing each remaining instance class must be re-calculated each time an instance class is tested because some of the elements in the PFS have just now been removed by performing that test. It took approximately 5 minutes using unoptimized PROLOG code on a SUN 3-60 to produce these first 30 test cases. That time could

be shortened considerably if the code were optimized.

In examining Table 8.2 one can see that after testing the first two traces we have called all the TOI functions at least once. RemoveElt, removeDups, and initBag are called in the first trace. AddElt is called in the second trace in addition to removeDups and initBag. Note that the trace that provides the most testing benefit is only the sixth trace to be selected. The reason trace 6 is not selected sooner is its high testing cost. There are six TOI function calls in that trace and each must be executed seven times.

To see how the specific tests arise for each instance class, we can examine trace #6 from Table 8.2. As we outlined in the previous section, we test a trace by applying each output function to that trace. For each of these combinations we determine all the different sub-instance classes, and produce an element from each sub-instance class that satisfies the appropriate constraints. Table 8.3 gives the different test cases for the sixth trace selected to be tested as outlined in Table 8.2.

Test Cases for D,D,D,E,A,I			
Output Function	Test Case	Correct Output	Constraints
size	size(γ)	1	i1 ≠ i2
size	size(γ)	0	i1 = i2
emptyBag	emptyBag(γ)	true	i1 = i2
emptyBag	emptyBag(γ)	false	i1 ≠ i2
includesElt	includesElt(γ,i3)	false	i1 = i2
includesElt	includesElt(γ,i3)	false	i1 ≠ i3
includesElt	includesElt(γ,i3)	true	i1 ≠ i2, & i1 = i3

Note: $\gamma = removeDups(removeDups(removeDups(removeElt(addElt(initBag(), i1), i2))))$

Table 8.3: Test Case Generation for Trace #6 in Table 8.2.

8.4. Discussion

We have applied the methodology outlined in the previous sections to ADTs with considerably more axioms in their semantics section. For example we used an extended list type called List-e which had 39 axioms, and more than one C-type operation. The specification of the type List-e, along with the PROLOG routines to determine the test set, and test cases produced for this large example are given in Appendix III. The results of that experiment produced test sequences similar to the one outlined in Table 8.2. That example is not included here for the

sake of brevity. Some of the more interesting and non-trivial operations of type List-e example are included in our shorter Bag-c example. These include the E-type functions "removeDups" and "removeElt" and the O-type function "includesElt." The outputs from these functions depend not only on the set of operations that generate their operands but also on the order in which those operations are applied, and on the non-TOI integer input values. The type List-e also included functions that accept more than one TOI value as input. Two such binary-TOI functions were intersection(list,list) and union(list,list), both of which return lists. The constraints on the sub-instance classes of these functions include the cartesian product of the constraints of the "input" lists. Although these functions lead to complex sub-instances and traces, they are handled within this methodology.

The most important application we have found for our computation space model of an ADT is in the ordering of test cases. This model of ADT computation may provide useful insight into problems other than testing such as software implementation and maintenance. For our research, using this model means that we do not need to assume that in a fault in the implementation of the ADT must occur in a single function. We allow for faults to be distributed over several functions. For example, an error might occur in how an "end of list" flag is set in one operation and then used in some other operations. Thus when examining possible faults we consider not only the functions that may be applied to a particular class of ele-

ment of the TOI (say queues of length four), but also the different combinations of ADT-operations and non-TOI values that may yield an element of that class.

Another important concept we have introduced in this chapter is the possible failure set (PFS). This concept provided two advantages. First, it gives us a sound basis to determine which instance class should be tested next. Secondly, by using a software testing goal of "making the implementation fail" and assuming limited resources to achieve that goal, we have been able to apply the concepts of PFS and equivalence space to produce a *sequence* of test cases. That is, we can say "run this test next." It is important to note that our choice of the "next test case" is not independent of what has already been tested. The number of PFS elements removed by testing a trace, and therefore the benefit of testing that trace, depends on which PFS elements have been removed by previous tests.

8.5. Validation

For the validation analysis of our test set generation method we use the RELAY model of error detection as presented in [Richardson88]. We selected this model because it is a well accepted model that builds on the idea that an error is created when an incorrect state is introduced at some fault location, and the error is propagated if it persists to the output. Richardson introduced the concepts of *"origination"* and *"transfer,"* as the first erroneous evaluation and the persistence of that erroneous evaluation, respectively. This fault-based model of testing relies on an

assumption that the module being tested bears a strong resemblance to some hypothetically-correct module. In Chapter 5 we showed that our method makes a similar assumption (Section 5.1).

Another reason the RELAY model is appropriate for our use is it allows that:

> "It is possible that the tested module produces correct output for all input despite a discrepancy between it and the hypothetically correct module" [Richardson88].

Since the point of using ADTs is to hide implementation details, it is important that any model we use allow that there be more than one way to implement a correct module.

According to the RELAY model; *given some potential fault,* a potential *error* originates if the smallest sub-expression in the block of code containing the fault evaluates incorrectly. We can view that block of code as existing in a *context* which contains the value of all variables. "A *context oracle* is a relation that relates an initial execution on a test to one or more acceptable contexts. Execution on one or more tests reveals a context error when the context is not acceptable by the oracle" [Richardson88]. A potential error in some expression transfers to a "super"-expression if the evaluation of the "super"-expression is also incorrect. Figure 8.8 is a representation of the RELAY model of error detection. The conditions necessary for a potential error to produce an output error is called the revealing condition for that error. In Figure 8.8 we can see that to produce an output error we need to:

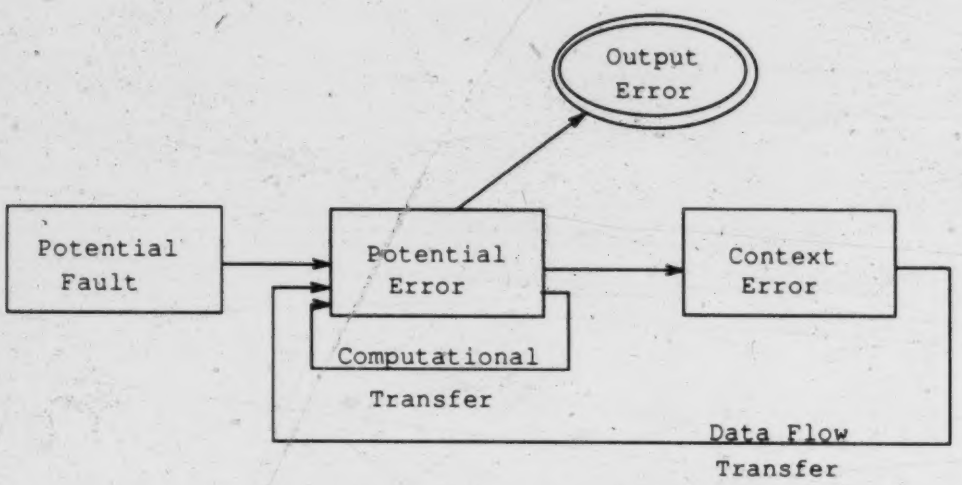


Figure 8.8 RELAY Model of Error Detection

(1) originate a potential error in the block of code with the potential fault;

(2) transfer that error through that block of code to produce a context error; this is called computational transfer;

(3) transfer that context error to the next block of code in the computation; this is called data-flow transfer;

(4) cycle through (2) and (3) until an output error is produced.

Therefore the revealing condition for a context error resulting from a potential fault f_n occurring at node n is the conjunction of the origination condition and the transfer conditions for f_n and n.

The RELAY model has been used for white box testing. Based on the generic revealing conditions outlined above, the model "is applied by first selecting potential faults for which a module is to be tested and instantiating the generic origination and transfer conditions to provide conditions specific to those faults" [Richardson88]. To apply the RELAY model in our case, we interpret the "module" where an error can happen as broadly as possible. We allow such a module to be as small as a single ADT function call, and to be as large as any series of ADT function calls.

Given that we allow a potential fault in any series of ADT-function calls, a potential error originates if the L(S) expression to be tested contains the (smallest) expression (say SEXP) such that

$$(SEXP) \neq (SEXP)^{\bullet}$$

for some set of constraints required by the specifications. Here $(SEXP)^{\bullet}$ is a hypothetically correct result for SEXP.

As long as all observable outputs of an inner expression E_i with constraints C_i are *active inputs* in the output of a "super"-expression E_s with constraints C_s containing E_i, any errors in the output of E_i will be propagated to E_s.

Definition (Active Input):

> Given a set of operations $\{op_1, op_2, op_3, \ldots, op_n\}$ an expression $E = I_1\, op_a\, I_2\, op_b\, I_3\, op_c \cdots$, I_x is an active input in E iff a change in the value of I_x will always cause a change in the value of E for all possible values of the other "I"s.

It is possible that an error in an inner expression E_i could be masked and not transferred to a super expression E_s. In such a case E_i does not satisfy our definition of "active input" for E_s. If E_i is an active input for E_s then the output of E_s will depend on the constraints for E_i. Therefore $C_i \subseteq C_s$. If E_s is always independent of E_i then a potential error in E_i may not propagate to E_s. In such a case all the constraints C_s for E_s will be independent of the constraints C_i for E_i. In this second case $C_i \not\subset C_s$. If E_s is sometimes independent of E_i depending on condition c_x such that:

$$E_s = \text{IF } c_x \text{ THEN } F_1(E_i) \text{ ELSE } F_2$$

then the constraints for E_s will be:

$$C_s = \{c_x \cup C_i,\ \overline{c_x}\}.$$

Thus in this third case $C_i \subseteq C_s$ and the potential error in E_i will be propagated to E_s in the $c_x = true$ case. Therefore if we require the constraints of an inner expression to be propagated outward, we cause any context error in the inner expression to transfer to the outer super-expression.

8.5.1. Example

To give an example of a fault that may or may not be propagated we will use our "Queue with Has" example as outlined in Chapter 2. The algebraic specification for that ADT is given in Figure 8.9.

Let us assume we have an error spread over the series of functions:

Addq(Addq(...) ...)

If we test the trace

$$T_1 = Addq(Addq(Addq(Newq(),1),2),3)$$

we see that

$$T_2 = Addq(Addq(Newq(),1),2)$$

is an inner trace or "sub-expression." The question now becomes "In testing T_1 will we reveal the potential error in T_2?"

When we apply the O-type functions (in this case Isnewq, Frontq, Has) to the inner trace T_2 we get the functionality presented in Table 8.4. When we apply all the O-type functions to T_1 we get the functionality presented in Table 8.5. We can see that:

-T1 implies E1
-T2 implies E2
-T4 implies E3
-E1, E2 and E3 are the only constraints for T_2.

Therefore testing T_1 will reveal our potential error.

Type Queue(Integer)

SYNTAX

Newq	-> Queue
Addq(Queue,Integer)	-> Queue
Deleteq(Queue)	-> Queue
Frontq(Queue)	-> Integer U {error}
Isnewq(Queue)	-> Boolean
Has(Queue,Integer)	-> Boolean

SEMANTICS

For all q : Queue; i ; Integer , Let

Isnewq(Newq)	- True
Isnewq(Addq(q,i))	- False
Deleteq(Newq)	- Newq
Deleteq(Addq(q,i))	- IF Isnewq(q) THEN Newq ELSE Addq(Deleteq(q),i)
Frontq(Newq)	- error
Frontq(Addq(q,i))	- IF Isnewq(q) THEN i ELSE Frontq(q)
Has (Newq,j)	-False
Has (Addq(q,i),j)	-IF i=j THEN True ELSE Has(q,j)

END Queue

Figure 8.9 Algebraic Specification of Queue with Has

Constraints for T_e			
Output Function	Correct Output	Constraints	Constraint #
Isnewq	false	none	...
Frontq	i1	none	...
Has(T_e,x)	true	x=i1	E1
Has(T_e,x)	true	x=i2	E2
Has(T_e,x)	false	x≠i1 & x≠i2	E3

Table 8.4: Observable Functionality for T_e

Constraints for T_t			
Output Function	Correct Output	Constraints	Constraint #
Isnewq	false	none	...
Frontq	i1	none	...
Has(T_t,x)	true	x=i1	T1
Has(T_t,x)	true	x=i2	T2
Has(T_t,x)	true	x=i3	T3
Has(T_t,x)	false	x≠i1 & x≠i2 & x≠i3	T4

Table 8.5: Observable Functionality for T_t

Now let us consider testing the trace

$$T_x = Deleteq(Addq(Addq(Newq(),i1),i2)).$$

We see that T_e is also a sub-expression of T_x. When we apply all O-type functions to T_x we get the functionality presented in Table 8.6. We can see that constraints E1 and E3 are not implied by any of the constraints arising from testing T_x. Thus the context error from T_e may not be transferred to T_x as our transfer condition has not been met. Therefore testing T_x may not reveal our potential error.

Constraints for T_2			
Output Function	Correct Output	Constraints	Constraint #
Isnewq	false	none	...
Frontq	i1	none	...
Has(T_2,x)	true	x=i2	X1
Has(T_1,x)	false	x≠i2	X3

Table 8.6: Observable Functionality for T_2

If we step back and look at this example from a more general perspective we can see that it makes sense. We may be able to say we have tested a queue of length 2 (T_2) after we have tested a queue of length 3 (T_1). It would be premature to say we have tested a queue of length 2 after we have tested a queue of length 1 (T_2). If we extrapolate this process of saying we have tested a queue when we have tested a shorter queue we could eventually say we have tested all queues of any length by testing an empty queue! Thus, from this example we see that this model also makes intuitive sense.

8.5.2. Model

For our analysis we will look at how many potential faults are *revealed* (as described earlier in this section) by our test set generation method and by the methods published previously [Bouge85b, Wild86, Choquet86, Bouge86]. Our error model (based on the RELAY model) is as follows:

Potential Faults

A potential fault may occur in any single ADT-function or may be spread over any possible series of ADT-functions.

Origination Condition

The test expression SEXP is the smallest expression containing a potential fault:

$$(f_1 \cdots f_n) \neq (f_1 \cdots f_n)^*$$

for some set of constraints required by the specification of the ADT.

Transfer Condition

$\forall f \in ADT$:

If the constraints required in

$f_1 \cdots f_n$

are all required in

$f_x f_1 \cdots f_n$

then a potential error in $f_1 \cdots f_n$ is transferred to $f_x f_1 \cdots f_n$

Figure 8.10: Validation Error Model

8.5.3. Experimental Procedure

In this sub-section we describe the experimental approach we used to compare our test set generation method with previously published methods [Bouge85b, Wild86, Choquet86, Bouge86]. For all experiments we used SICStus PROLOG.

We used the following experimental procedure for the example ADTs used in this thesis (Queue with Has, specified in Figure 8.9; Stack (with a hidden function), specified in Figure 3.7; and Bag-c, specified in Figure 8.6).

Given an ADT and its algebraic specification:

1) Produce an old-style PROLOG specification using the equivalences given in Section 3.2.3.

2) Produce a PROLOG specification using our new equivalences given in Figure 6.6.

3) REPEAT until one method reveals 300 potential faults,

3.1) Generate the next set of test cases according to our method as outlined in this chapter, using the PROLOG specifications produced in step 2).

3.2) Determine all possible faults that the tests from step 3.1) will reveal (using the model given in the previous sub-section).

3.3) Determine which of the faults from step 3.2) have not already been revealed by tests previously generated by our method and add them to the count of "faults revealed" by our method.

3.4) Generate the next set of test cases according to the methods described by previous authors, using the PROLOG specifications produced in step 1).

3.5) Determine all possible faults that the tests from step 3.4) will reveal.

3.6) Determine which of the faults from step 3.5) have not already been revealed by tests previously generated in step 3.4), and add them to the count of "faults revealed" by previous methods.

4) Produce plots of "faults revealed" for each method versus the number of test cases run and versus the number ADT function calls executed.

Procedure 8.1: Experiment for Comparing ADT Testing Methods

The "possible faults revealed" (steps 3.2 and 3.5) were determined by exhaustively looking at all operation sentences in the traces of the test cases that have been generated and then determining which of them gave potential faults that would be revealed according to the RELAY model of error detection.

300 faults was selected as a stopping point because that was how far the experiment had progressed after running for 24-48 hours (on a MIPS M1000 with 32 M-bytes of memory and a load average below 2.00), and because the plots of the results appeared to be stable.

8.5.4. Results

Figures 8.11- 8.16 are graphs of "potential faults revealed vs. number of test cases run," and "potential faults revealed vs. number of ADT function calls" generated by the procedure described in the previous sub-section for the example ADTs we have used in this thesis.

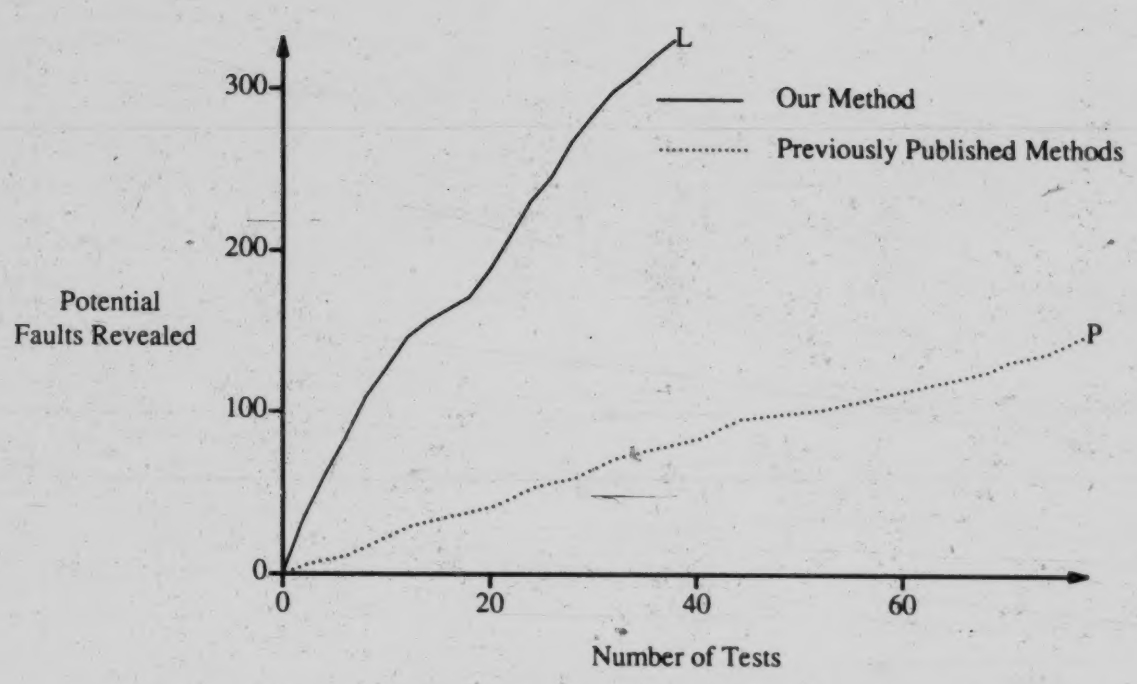


Figure 8.11: Potential Faults Revealed vs. # Tests Run. For Type: STACK

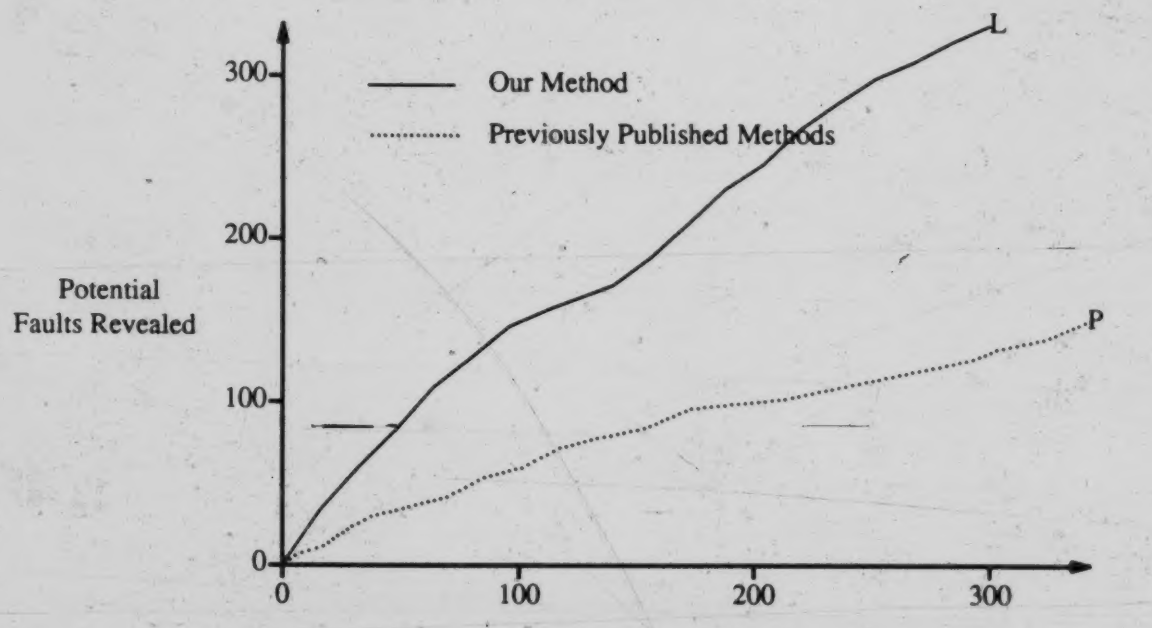


Figure 8.12: Potential Faults Revealed vs. # ADT Function Calls. For type: STACK

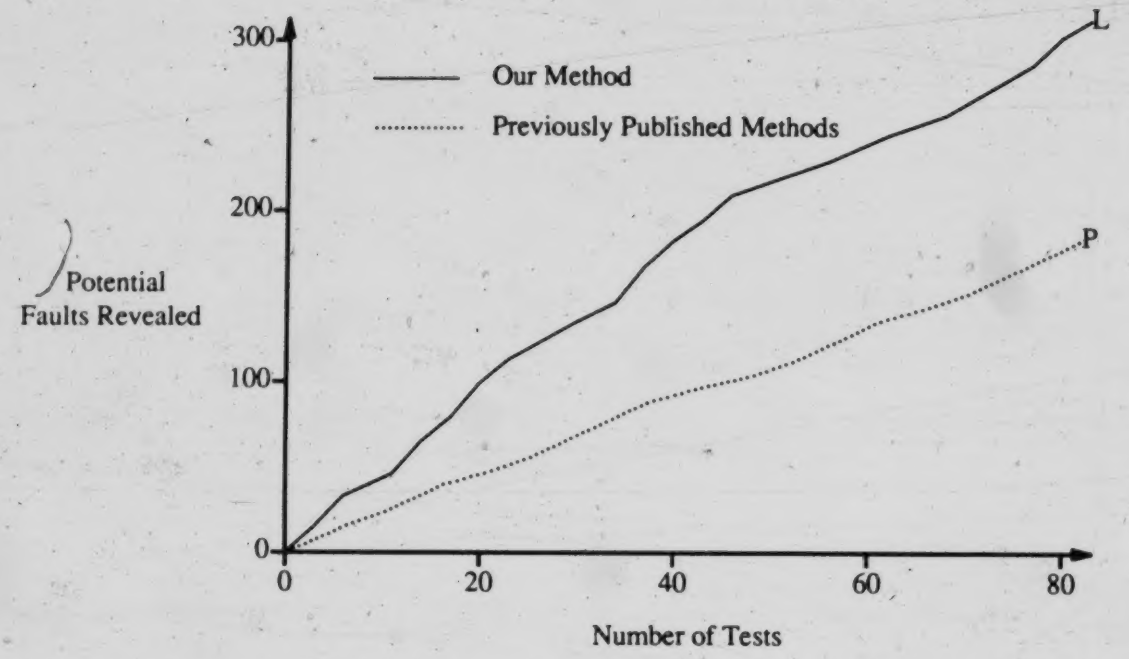


Figure 8.13: Potential Faults Revealed vs. #Tests Run. For type: QUEUE WITH HAS

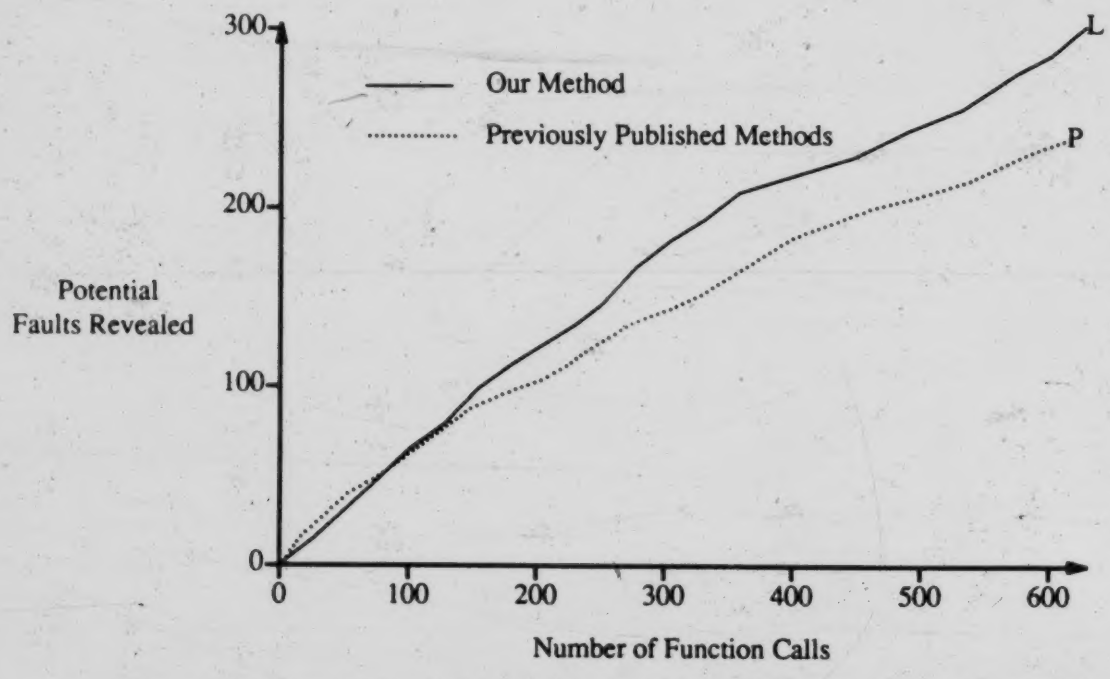


Figure 8.14: Potential Faults Revealed vs. # ADT Function Calls. For Type: QUEUE WITH HAS

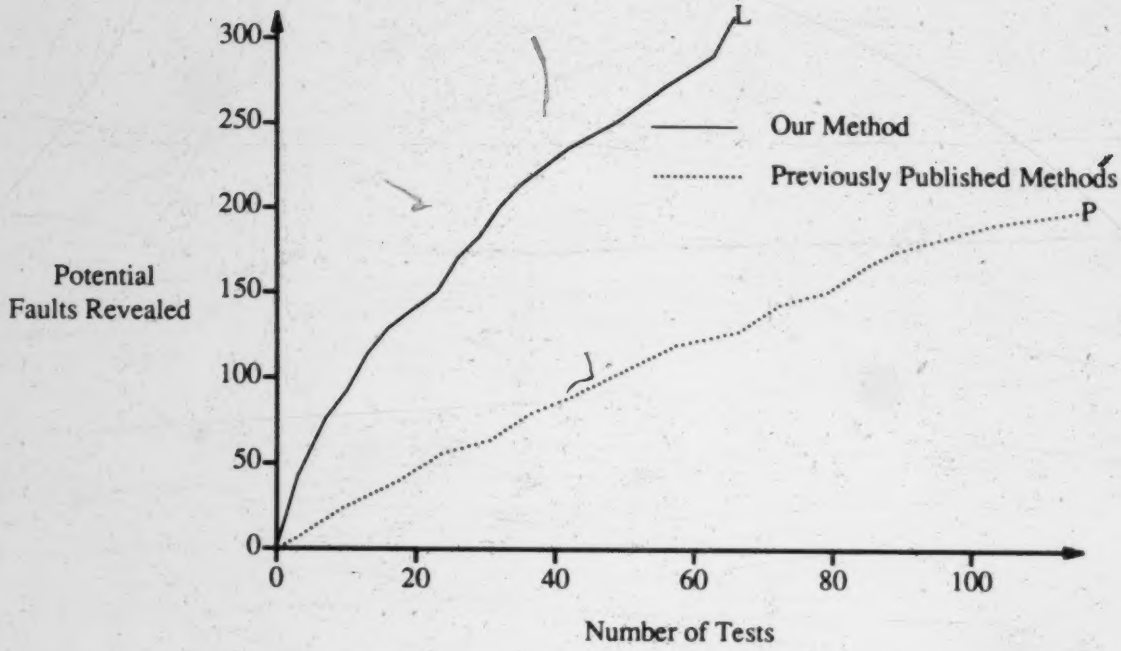


Figure 8.15: Potential Faults Revealed vs. # Tests Run. For Type: BAG.

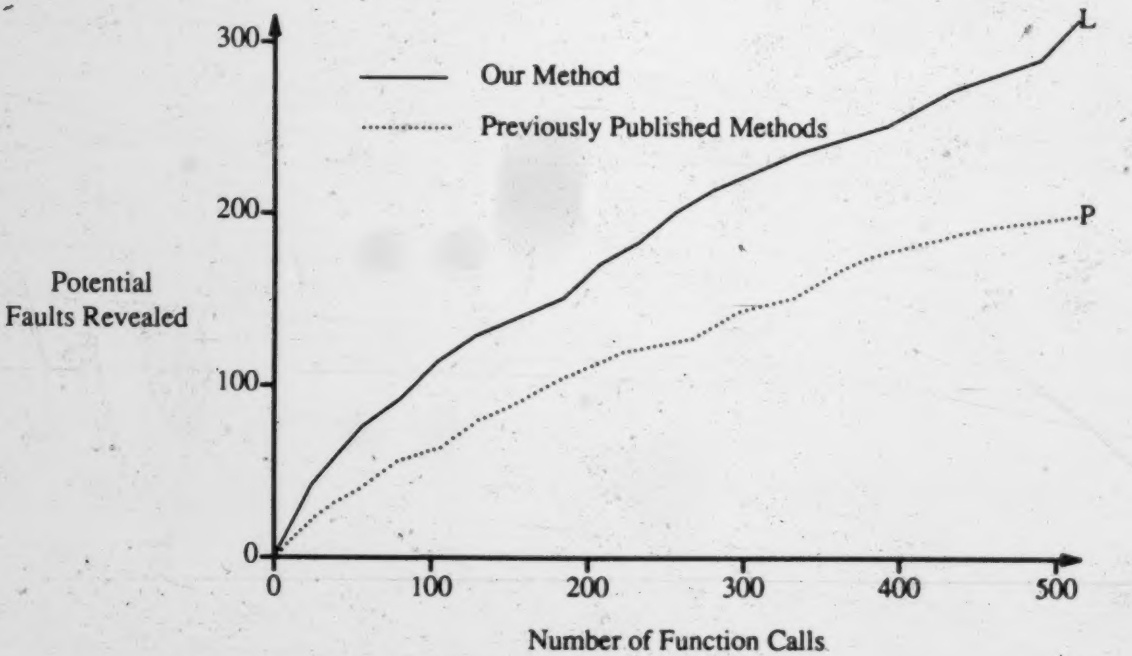


Figure 8.16: Potential Faults Revealed vs. # ADT Function Calls. For Type: BAG.

8.5.5. Analysis

Using the nonparametric sign test for paired comparisons [Chapman70], we can accept at better than the 99 percent confidence level that our new method reveals more potential faults than the previous method. We note that this experiment does not require any correlation of faults. The experiment views the faults revealed as independent.

There are two basic factors that caused our method to reveal more potential faults than the previously published methods. The first factor is that our method does not assume "The implementation of all E-type functions is correct." In Section 4.5.1 we showed that previous methods do make that assumption. Thus as the number of tests or the number of ADT function calls approach infinity, there will be many potential faults that are not tested by the previous methods but are profitably tested by our method. As our example in Section 4.5.1 showed, the potential faults that previous methods did not reveal can exist in even very short traces. This problem is described in detail in Chapter 4.

The second factor that caused our method to reveal more potential faults than the previously published methods is the fact that the previous methods start with the shortest possible trace and test all shorter traces before longer ones. For simple ADTs that are not affected by their input values and have short specifications with no "IFs" (QUEUE WITH HAS is simple in this sense), testing almost becomes a problem of

pattern matching. Since these simple ADTs have little observable functionality, all traces with the same ADT functions will act the same and the tester need only ask "have I tested this pattern of ADT function calls yet." As we introduce more functionality (Figures 8.15, 8.16, 8.11, 8.12), some traces will be better than others for testing purposes. In that case, requiring all shorter traces to be tested first becomes more and more expensive, because the shorter traces may not *necessarily* be as good as longer traces. Our method is not restricted to testing all the shorter traces first. Therefore as the ADT to be tested has more and more observable functionality, our method outperforms the previously published methods by larger and larger margins. This has been borne out in our results. QUEUE WITH HAS (Figures 8.13, 8.14) is less complex than BAG (Figures 8.15, 8.16), which is less complex than STACK (Figures 8.11, 8.12). In all cases our method reveals more faults than previous methods. When more complex ADTs are produced and require testing, our method can be expected to outperform previously published methods by larger and larger margins.

9. SUMMARY AND CONCLUSION

A great many industrial, government, and academic concerns are making large investments in software systems whose successful operation is critical to their on-going operations. Today, the failure of a software system could possibly be measured in human lives in the case of the American space shuttle, in hundreds of lives in the case of the AirBus A-300 airliner, or in a catastrophic situation, thousands of human lives in the case of a CANDU nuclear reactor. Therefore the ability to produce reliable software is currently of great importance. Unfortunately, the problem of determining if a program obeys its specifications is undecidable in general. Therefore the production of software systems that are as reliable as possible requires the use of *several* validation and verification techniques. These include system and code review, software inspection, and more than one type of testing. These various software validation and verification techniques do not exist in isolation, but co-exist and complement each other in improving software reliability, safety, and correctness. This thesis presents a new testing technique that is just now becoming feasible. This technique will help insure that software "does everything it should," and thus in co-existence with other validation and verification techniques will help to increase the reliability of software systems. Developing this technique required several research contributions. These include:

(1) building a prototypical system based on the combination and extension of previous research results;

(2) developing a foundation for a general method for specification directed testing;

(3) developing a new method for using PROLOG to generate test cases;

(4) developing a computational model for data abstraction;

(5) developing a description, called the possible failure set (PFS), of the ways an abstract data type can fail;

(6) comparing our technique to previously published techniques.

9.1. Summary

The initial focus of our work was the development of a new specification-directed software testing methodology, and the construction of a prototype called T-3. T-3 is a combination and extension of previous research results in test harness construction and test case generation using PROLOG. T-3 can test a limited class of ADTs against their specifications and only requires the specifications and the implementation as input. This system is a significant advance as it solves the oracle problem. Most software testing methodologies previously described in the literature need or have assumed the existence of an entity or oracle that could determine if the output from a software implementation was correct. Our methodology does not need any external entities to determine the correctness of test results. The limitations

on the ADTs that can be tested by T-3 are listed in Section 3.4. These limitations result from limitations in previous test harness construction methods and previous black-box test case generation methods. The main focus of our work has been to determine and solve the underlying problems that led to these limitations.

Since ADTs can be viewed as running on an abstract machine, a specification directed testing methodology for ADTs should be applied in a bottom-up manner. This allows us to assume the correctness for lower types. That is, we assume the abstract machine we are working on is correct.

Given the abstract machine we are using is correct, we developed a new concept of "correct" for an ADT implementation. We did this by separating the predicate 'OK' into two parts: OK_b and OK_w. Now for our black-box testing purposes, a correct implementation I of an ADT T= (V,S) is one for which every O-type function that is a member of V returns correct (OK_b) values for every element of its domain. For black-box testing we are only interested in the OK_b portion of the OK predicate. This definition of "correct" allowed us to state the goal of our overall testing system. That goal is "to generate and observe a software implementation failure as quickly as possible." Instead of selecting tests such that "the more tests we successfully run, the more likely the software is correct," we select tests such that "the more (ADT) operations we execute the more likely we are to detect failures."

We have defined what we have found to be the necessary properties of a *"valid complexity metric."* Given our overall testing goal and the concept of correctness for an ADT, we showed that a testing methodology that iteratively tests instance classes in an order dictated by a valid complexity metric will produce an acceptable collection of tests.

We have shown how dangerous it can be to not explicitly state the assumptions intrinsic to a software testing methodology. We have investigated the assumptions and constraints on the domain of software to be tested by our specification-directed software testing methodology. Those assumptions and limitations are listed in Section 5.3.5.

A new test case generation method using PROLOG has been developed. This method rests on the new general theoretical method we have developed for achieving our testing goal. This method generates test cases for a given instance class. The two key tasks for testing an instance class are: determining all the sub-instance classes, and determining all the expected results for each sub-instance class. To generate test cases an algebraic specification is translated into horn clauses using the equivalences in Figure 6.6. These horn classes are used as a PROLOG program. That program is used to find all the solutions to goals formed by applying an O-type function to an instance class. Each solution to those goals becomes a test. This PROLOG method provides three significant improvements over previous methods.

(1) We can handle general predicates in a uniform method as opposed to only allowing equality predicates.

(2) We can now test all functions of the ADT *and* all their combinations.

(3) We do not repeatedly produce the same test case over and over.

These improvements significantly expand the class of ADTs that we can test using PROLOG and reduce the assumptions we need to make about their implementation.

A new computational model for ADTs was developed to give some insight into what "testing" an element of an ADT means. This model was used to determine an ordering of the elements of a test set. The concept of a canonical form for any element of an ADT was developed within this model and was used to define two new entities: the equivalence spaces and the computation space of an ADT. Equivalence spaces partition the space of all elements of the TOI such that every member of an equivalence space has the same canonical form. In this model all C-type functions and E-type functions can be viewed as mappings from one equivalence space to another within the same *computation space*. These mappings and their associated equivalence spaces make up the computation space of the ADT. N-type functions are mappings from outside the computation space to an equivalence space within that computation space. An O-type function is a mapping from an equivalence space to a value outside the ADT's computation space.

According to this model of the computations of an ADT, when an element of the TOI is "tested" we have checked:

(1) that it has been mapped correctly into its appropriate equivalence space;

(2) that the observable functionality of *that* equivalence space is correct.

This model of ADT computation allowed us to model the conditions for an ADT failure. Those conditions are such that there must exist at least one equivalence space such that either:

(1) one of the traces that lead to it contains a fault; or

(2) one of the mappings from it (to an element outside the computation space) contains a fault.

Note that these two conditions correspond to the two problems we have checked when an element of the TOI is tested. Thus the Possible Failure Set (PFS), which contains all the ways an ADT can fail, is a union of two sets of conditions. Therefore it can be viewed as the union of

(1) the set of different traces to each equivalence space; with

(2) the set of output function mappings from each equivalence space.

Testing can be viewed as removing elements from the PFS. This leads to a "valid complexity metric" with which to order the instance classes to be tested. That

metric is used in the new algorithm for testing an ADT that was presented in Figure 8.5. The algorithm for selecting the next instance class to test is to select the next instance class with the best benefit/cost ratio as determined by the equation in step 2) of Figure 8.5. It should be noted that if we view this as a benefit/cost metric; the benefit of testing is the removal of elements from the PFS, and the cost of testing is the number of ADT function calls required.

A small demonstration of the overall methodology is given in Section 8.3. Appendix III gives the test set that this new methodology produces for a large complex example.

A major advantage of our methodology is that we allow the user to use the hierarchical nature of ADTs. We have shown in Section 2.3.1 that previous methods [Bouge85b, Choquet86, Wild86, Bouge86] for using PROLOG to produce test cases required O-type functions to return values of a type with an equivalence function. Therefore ADTs could only be built on types that had equivalence functions. That is a severe restriction. In Section 6.3.2 we showed the our PROLOG methodology does not require the values returned by O-type functions to have an equivalence function defined on them. Thus a user can build an ADT on top of another ADT that does not have an equivalence function.

Experiments have shown that our methodology revealed more faults than previously published methods. These experiments were run on ADTs for which the

previous methods work. We showed in Chapters 3 and 6 that there are a great many ADTs which the previous methods simply cannot test but which our methodology can.

Previously published methods for ADT testing have been based solely on the syntax of the ADT's specifications. We found it necessary to look at "what the ADT was actually computing." The ADT computation model presented in Chapter 7 allows us to do that. We have shown that such a method will reveal more software faults.

9.2. Conclusion

In Chapter 1 we stated that the initial hypothesis of our research has been that a complete testing methodology for abstract data types that is based on software specifications could be developed. We can now accept that hypothesis. A complete software testing system, called T-3, based on software specifications has been developed. We have extended the methodology that T-3 implements to handle a wider range of software and to produce fundamentally sound and usable test sets. In Section 8.5 we have shown that this methodology reveals more faults than previously published methods.

9.3. Future Work

One of the main contributions of our work is the new testing algorithm given in Figure 8.5. Step 2) of that algorithm requires the user to determine which untested

instance class has the largest benefit/cost value. This is currently done by exhaustively searching all traces of length one followed by all traces of length two, and so on. This leads to some very long, slow, searches for the more complicated ADTs such as the one in Appendix III. Algorithms for pruning and shrinking that search space need to be developed. These algorithms should be based on the syntactic structure of the traces examined, and could also be based on our computational model for ADTs. In Section 8.5 we showed that it took 24-48 hours on a MIPS M1000 to reveal 300 possible faults. We believe that time could be shortened by one or two orders of magnitude by effective trace-search algorithms.

Our overall testing methodology partitions an input domain into instance classes and selects the best untested instance class. To test an instance class we test all its sub-instance classes. It may be the case that some of these sub-instance classes tend to be more important to the testing process than others. This importance may be dependent on how the software was produced, who produced it, or how it is to be used. An algorithm for ranking sub-instance classes may produce more efficient test sets.

Our testing method selects the next instance class to be tested based on the value of benefit/cost. This metric can also be viewed as "number of possible failure space elements removed per function call." This method has been shown to be a significant improvement over previous methods. The benefit/cost metric we use is not

the only selection metric possible. For more complicated ADTs or as computer hardware gets faster it may be more useful to use different metrics for benefit and cost. An investigation of revised benefit/cost metrics for our overall testing methodology, and when to use them, would be useful.

ADTs can be viewed as running on "abstract machines" that provide all the types and operations necessary to conveniently specify and build the ADT. In Section 5.2 we showed that our methodology assumes the correctness of the abstract machine on which the ADT runs. An interesting extension of our work would be to develop a methodology that relaxes that assumption. This would require an investigation of:

(1) the effect of faults in lower types on the testing of higher types; and

(2) the possibility of testing lower types while testing a particular "higher" ADT.

Such an investigation would lead to a method for developing an integrated test plan, for specification-directed testing of several ADTs.

Our overall testing methodology requires an algebraic specification of the ADT. The main reason we cannot use model oriented specification techniques is there are no logic programming facilities available yet that are based on such objects as sets and sequences. When such facilities are available an investigation of extending our methodology to model oriented specifications would be valuable. That investigation will have to develop ways of handling the existential quantification and side effects

that are allowed in model oriented specifications and not allowed in algebraic techniques.

The equivalences developed in Chapter 6 that produce a PROLOG version of an ADT's specification do not produce rules that lend themselves directly to parallel implementation. Each clause in these rules uses the "constraints_out" of the previous clause as its "constraints_in." This usually means these clauses cannot be solved in parallel. A PROLOG specification that effectively implements our test case generation method for a parallel-PROLOG system could be developed by looking at different ways of implementing the "add_constraints" clause in equivalence 1) of Figure 6.6. This could significantly increase the efficiency of the overall testing methodology.

T-3 is only a prototype; a production quality system based on the T-3 model needs to be built. The next logical step for the test case generation methodology outlined in this thesis will be to integrate an implementation of our new test set generation methodology into a complete testing tool similar to T-3. While the T-3 system used a rather simple trace generation algorithm to generate the test cases, it did demonstrate that our general methodology can be used to produce a useful working system. We have seen in a large example that by using the test case generation methodology outlined in this paper, such a system will not require infinite resources to test an ADT implementation. In fact it will exercise it in a

useful and practical manner.

Now that a foundation for specification-directed software testing has been set down and a methodology developed, it would be useful to investigate what types of faults are revealed in the implementation code by different testing strategies. A useful study would be one that required programmers with different skill levels to implement several different ADTs, and then test those ADTs and any earlier "working copies" that might have been produced with our testing method. Such a study would produce some guidelines for determining when our methodology would be most useful and when other validation and verification methods would be more appropriate.

Our testing methodology is not intended to exist in isolation. The production of software systems that are as reliable as possible requires the use of several validation and verification techniques. We have shown that a specification-directed software testing methodology such as ours can only address the "OK_b" half of the testing problem. We also showed that white box testing methods could only address the "OK_w" half of the problem. An investigation of the advantages and problems of combining white-box and black-box testing methods is needed. The hierarchy of white-box coverage measures is already well defined. As a first step toward combining white-box and black-box techniques, a two or three dimensional hierarchy that reconciles white and black-box coverage measures must be developed.

Abstract data types have been described as "half way to an object" (in the object oriented programming sense)[Ingalls89]. Therefore, we have developed a solution for part of the problem of software testing in an object oriented environment. We have looked at encapsulation. A very interesting and useful extension of our results would be the design and construction of a testing system for an object-oriented programming environment. That system should be based on T-3 and extended to handle inheritance in object oriented environments. To build such a system research must be done to determine how and when faults are transmitted through an inheritance hierarchy in an object oriented environment. It will be particularly important to characterize when faults may be transmitted and when faults are guaranteed to be transmitted through an inheritance hierarchy.

GLOSSARY

Acceptable Battery of Tests
A battery of tests T for a testing context C is acceptable if it is asymptotically valid and unbiased.

ADT
Abstract Data Type. An abstract data type defines a class of abstract objects that is completely characterized by the operations available on those objects.

Asymptotic Validity
Let $C = \langle L, S, (\Pi), A \rangle$ be a testing context, and $T = \langle H, (T_n)_{n \in N} \rangle$ a battery of tests for that context. T is asymptotically valid if for every L(S)-structure Π of (Π) if $\Pi \models T_n$ for every $n \in N$ then $\Pi \models A$.

Carrier
The carrier of an algebra is the set of mathematical objects we wish to manipulate with that algebra. [Stanat77] Examples might be integers, real numbers, or a set of character strings.

Complexity Metric
A complexity metric can be any function that can be applied to all possible test cases and returns a numeric result.

Context
See testing context.

Deterministic
Denoting a method, process etc., the resulting effect of which is entirely determined by the inputs and initial state.

Error
An error is a piece of information which, when processed by a system, may produce a failure.

Experiment
Let $C = \langle L, S, (\Pi), A \rangle$ be a testing context. An experiment E for C is a L(S)-formula without quantifier, such that for any non-logical symbol p of E and for any L(S)-structure Π of (Π), the meaning of p in Π is calculable.

Failure — A failure for a software system is an observable event where the system violates its specification.

Fault — A fault is an algorithmic or mechanical defect which may generate an error.

Functionality — Let i be an instance of a language L(S), and a L(S)-structure Π be an implementation. The functionality of i in Π is the set of pairs $\langle o_n \in O , v_n \rangle$, where O is the set of O-type operations in L that may be applied to i such that their meaning in Π is calculable, and v_n is that meaning in Π.

Homomorphism — A structure preserving mapping between algebras. Let G and H be two algebraic structures of the same type in the sense that G has a binary operation $\circ$ and H has a binary operation $\cdot$ defined. Then

$$\phi : G \rightarrow H$$

is a homomorphism (homomorphic mapping) provided it is a function from G into H and

$$\phi(g_1 \circ g_2) = \phi(g_1) \cdot \phi(g_2)$$

for all g_1 and g_2 in G. [Illingworth83]

Instance — Let L be language and S a set. An instance i of L(S) is an L(S)-formula without quantifier or logical symbols whose symbols are $p_1, p_2, p_3, \ldots, p_n$ where n is the number of L(S) symbols in i.

Instance Class — Let L be a language and S a set. An instance class I of L(S) is formed from an instance $i = p_1, p_2, p_3, \ldots, p_n$ of L(S) by including all instances $i' = p'_1, p'_2, p'_3, \ldots, p'_n$ of L(S) that have the same number of symbols as i and where if p_x is different from p'_x then $p_x \in S$ and $p'_x \in S$. That is, an instance class contains all instances with the same trace of functions.

Language — Any set of strings over an alphabet Σ, that is, any subset of Σ^* is called a Σ- language. [Lewis81]

L(S)-structure

In this thesis an L(S)-structure may be viewed as a program or an implementation. Formally: "a structure is a pair P=([P],I_p), where [P] is any non-empty set called the universe of P and I_p is a function having as its domain a set of predicate and function signs. Specifically,

(1) if Q is an n-place predicate sign in the domain of I_p, then I_p(Q) is an n-ary relation on [P], that is, a subset of $[P]^n$;

(2) if f is an n-place function sign in the domain of I_p, then $I_p(f)$ is a function from $[P]^n$ to $[P]$." [Lewis81]

L(S)-theory

In this thesis the set of axioms in an algebraic specification of an ADT can be viewed as a theory about that ADT. Formally, a theory T on a language L is any set of L-formulas.

Nondeterminism

A mode of computation in which, at certain points, there is a choice of ways to proceed: the computation may be thought of as choosing arbitrarily between them or as splitting into separate copies and pursuing all choices simultaneously. Nondeterminism is important in the field of complexity: it is believed that a nondeterministic Turing machine is capable of performing in "reasonable time" computations that could not be so performed by any deterministic Turing machine.

Observable Functionality

Let L(S) be a language, let an L(S)-structure Π be an implementation, and let T be a set of instances of L(S). The observable functionality of Π for T is the union of the functionalities of all instances $i \in T$ in Π.

Operation Sentence

An operation sentence for an ADT is:

Base Case: a application of a single operation for that ADT;

or

Recursive Case: an application of a operation of that ADT to an operation sentence of that ADT.

Projective Reliability Let C = $\langle L, S, (\Pi), A \rangle$ be a testing context, and $(T_n)_{n \in N}$ a countable family of tests for that context. $(T_n)_{n \in N}$ is projectively reliable if for every $n \in N$ and for every structure Π of (Π) such that $\Pi \models T_{n+1}$ then $\Pi \models T_n$.

Reliable Test Criterion

Given program domain D a test criterion C and two test sets T_1 and T_2, the criterion C is reliable if:
$Reliable(C) = (\forall T_1, T_2 \subseteq D)\,[\,(C(T_1) \wedge C(T_2)) \rightarrow (Successful(T_1) \leftrightarrow Successful(T_2))\,]$

Sub-Instance Class Let L be a language, S be a set, I be an instance class of L(S), and A be an L(S)-theory. For an instance $i = p_1, p_2, p_3, \ldots, p_n \in I$ with input values $q_1, q_2, q_3, \ldots, q_m$, there exists a function F_i such that according to A $F_i(q_1, q_2, q_3, \ldots, q_m) \equiv i$. A sub-instance class I_i of I is formed from i by including all instances $i_x \in I$ with input values $q_{x1}, q_{x2}, q_{x3}, \ldots, q_{xm}$ such that according to A $F_i(q_{x1}, q_{x2}, q_{x3}, \ldots, q_{xm}) \equiv i_x$.

Testing Context A testing context C is a 4-uple $\langle L, S, (\Pi), A \rangle$ where L is a first order language; S is a set; (Π) is a family of L(S)-structures; and A is an $L'(S)$-theory. Where $L \subseteq L'$.

Trace The sequence of ADT-function applications which generated a data object from some initial state is the trace of that data object.

Test Let C = $\langle L, S, (\Pi), A \rangle$ be a testing context. A test T for C is an L(S)-theory with only a finite number of axioms, each of them being an experiment of C.

Type of Interest For an ADT, the data type defined solely in terms of other types is called the Type of Interest (TOI).

Unbiased A test set is said to be unbiased if whenever the implementation to be tested is correct it will pass that test.

Uniformity Hypothesis

A sub-domain is uniform with respect to correctness if there

exists an element of that sub-domain such that its correctness implies the correctness of all elements of that sub-domain.

Valid Test Criterion Given program domain D and a test set T, a test criterion C is said to be valid if:
$Valid(C) = (\forall d \in D)[\neg OK(d) \rightarrow (\exists T \subseteq D)(C(T) \wedge \neg Successful(T))]$.

REFERENCES

[Abramson84] H. Abramson, "Definite Clause Translation Grammers," 1984 Intl. Symposium on Logic Programming, IEEE Comp. Soc. Press, pp. 233-240, Feb. 1984.

[Adrion82] W.R. Adrion, M.A. Branstad, J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," ACM Computing Surveys, vol. 14, no. 2, pp. 159-192, 1982.

[Berg82] H.K. Berg, W. Boebert, W. Franta, T. Moher, "Formal Methods of Program Verification and Specification," Prentice Hall, 1982.

[Berztiss83] A.T. Berztiss, S. Thatte, "Specification and Implementation of Abstract Data Types," Advances in Computers, Academic Press, vol. 22, pp. 295-355, 1983.

[Bouge85a] L. Bouge, "A Contribution to the Theory of Program Testing," Theoretical Computer Science, vol. 37, pp. 151-181, 1985.

[Bouge85b] L. Bouge, N. Choquet, L. Fribourg, M.C. Gaudel, "Application of PROLOG to Test Sets Generation From Algebraic Specifications," TAPSOFT Joint Conference on Theory and Practice of Software Development, vol 2, pp. 261-275, Berlin, March, 1985.

[Bouge86] L. Bouge, N. Choquet, L. Fribourg, M.C. Gaudel, "Test Sets Generation From Algebraic Specifications Using Logic Programming," Software and Systems vol. 6, no. 4, pp. 343-360, 1986.

[Budd78] T.A. Budd, R. DeMillo, R.J. Lipton, F.G. Sayward, "The Design of a Prototype Mutation System for Program Testing," Proc. ACM Natl. Comp. Conf., pp. 623-627, 1978.

[Budd81] T.A. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," Computer Program Testing, North-Holland Publishing, pp. 129-148, 1981.

[Choquet86] N. Choquet, "Test Data Generation Using a Prolog with Constraints," Proc. ACM-IEEE Workshop on Software Testing, pp. 132-141, July, 1986.

[Davis85] R.E. Davis, "Logic Programming and Prolog: A Tutorial," IEEE Software, vol. 2, no. 5, pp. 53-62, Sept., 1985.

[Day85] J.D. Day, J.D. Gannon,"A Test Oracle Based on Formal Specifications," Proc. SOFTFAIR II, pp. 126-130, Dec., 1985.

[Feather82] M.S. Feather, "Program Specification Applied to a Text Formatter," IEEE TSE, vol. 8, no.5, pp. 490-498, Sept. 1982.

[Ford85] R. Ford, K. Miller, "Abstract Data Type Development and Implementation: An Example," IEEE TSE, vol. 11, no. 10, pp. 1033-1037, Oct. 1985.

[Gannon81] J.D. Gannon, P. McMullin, R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing," ACM TOPLAS, vol. 3, no. 3, pp. 211-223, July, 1981.

[Gannon87] J.D. Gannon, R.G. Hamlet, H.D. Mills, "Theory of Modules," IEEE TSE, vol. 13, no. 7, pp. 820-829, July, 1987.

[Gaudel88] M.C. Gaudel, B. Marre, "Algebraic Specifications and Software Testing: Theory and Application," Rapport de Recherche no. 407, Universite de Paris-Sud, Feb., 1988.

[Goodenough75] J.B. Goodenough, S.L. Gerhart, "Toward a Theory of Test Data Selection," IEEE TSE, vol. 1, no. 2, pp. 156-173, June 1975.

[Gougen77] J.A. Gougen, J.W. Thatcher, E.G. Wagner, J.B. Wright, "Initial Algebra Semantics and Continuous Algebras," JACM, vol. 24,

no. 1, pp. 68-95, Jan., 1977.

[Gougen78] J.A. Gougen, J.W. Thatcher, E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," Current Trends in Programming Methodology, vol. IV, pp. 80-149, Prentice Hall, 1978.

[Gourlay83] J.S. Gourlay, "A Mathematical Framework for the Investigation of Testing," IEEE TSE, vol. 9, no. 6, pp. 686-709, Nov. 1983.

[Guttag77] J. Guttag, "Abstract Data Types and the Development of Data Structures," CACM, vol. 20, no. 6, pp. 396-404, June, 1977.

[Guttag78a] J.V. Guttag, J.J. Horning, "The Algebraic Specification of Abstract Data Types," Acta Infomatica, vol. 10, pp. 27-52, 1978.

[Guttag78b] J.V. Guttag, E. Horowitz, D.R. Musser, "Abstract Data Types and Software Validation," CACM, vol. 21, no. 12, pp. 1048-1064, Dec., 1978.

[Guttag80] J. Guttag, "Notes on Type Abstraction (Version 2)," IEEE TSE, vol. 6, no. 1, pp. 13-23, Jan., 1980.

[Guttag85] J. Guttag, J. Horning, J. Wing, "The Larch Family of Specification Languages," IEEE Software, vol. 2, no. 5, pp. 24-36, Sept. 1985.

[Hayes86] I.J. Hayes, "Specification Directed Module Testing," IEEE TSE, vol. 12, no. 1, pp. 124-133, Jan., 1986.

[Heninger80] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," IEEE TSE, vol. 6, no. 1, pp. 2-12, Jan. 1980.

[Hoare72] C.A.R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, vol. 1, pp. 271-281, 1972.

[Howden76] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE TSE, vol. 2, no. 3, pp. 208-215, Sept., 1976.

[Howden80] W.E. Howden, "Functional Program Testing," IEEE TSE, vol. 6, no. 2, pp. 162-169, March 1980.

[Howden82] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," IEEE TSE, vol. 8, no. 4, pp. 371-379, July 1982.

[Howden85] W.E. Howden, "The Theory and Practice of Functional Testing," IEEE Software, vol. 2, no. 5, Sept. 1985, pp. 6-17.

[Howden86] W.E. Howden, "A Functional Approach to Program Testing and Analysis," IEEE TSE, vol. 12, no. 10, pp. 997-1005, Oct., 1986.

[Illingworth83] "Dictionary of Computing," V. Illingworth editor, Oxford University Press, 1983.

[Ingalls89] D. Ingalls, "Object Oriented Programming," Video Tape, Distinguished Lecture Series, vol. III, University Video Communications, 1989.

[Jaffar87a] J. Jaffar, J. Lassez, "Constraint Logic Programming," Proc. 14th. ACM POPL Conf. Munich, Jan. 1987, pp. 111-119.

[Jaffar87b] J. Jaffar, S. Michaylov, "Methodology and Implementation of a CLP System," Proc. 4th. Intl. Conf. on Logic Programming, Melbourne Austrailia, May 1987, pp. 196-218

[Kemmerer85] R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," IEEE TSE, vol. 11, no. 1, pp. 32-43, Jan., 1985.

[Lamb88] D.A. Lamb, "Software Engineering: Planning for change," Prentice Hall, 1988.

[Laski88] J. Laski, "Testing in Top-Down Program Development," Proc. Second Workshop on Software Testing, Verification and Analysis, Banff, Canada, pp. 72-79 July, 1988.

[Lawrynuik87] W.D. Lawrynuik, "The T-3 Testing Tool," Proc. Intelligence Integration, CIPS Edmonton '87, pp. 355-360, Nov., 1987.

[Lawrynuik89] W.D. Lawrynuik, L.J. White, "Test Case Generation for Specification Directed Testing of Abstract Data Types," Tech. Rept. TR89-22, University of Alberta Department of Computing Science, Aug. 1989.

[Lewis81] H.R. Lewis, C.H. Papadimitriou, "Elements of the Theory of Computation," Prentice-Hall, 1981.

[Liskov74] B. Liskov, S. Zilles, "Programming with Abstract Data Types," SigPlan Notices, vol. 9, No. 4, pp. 50-59, 1974.

[Liskov75] B.H. Liskov, S.N. Zilles, "Specification Techniques for Data Abstractions," IEEE TSE, vol. 1, no. 1, pp. 7-19, March 1975.

[Liskov77] B.H. Liskov, V. Berzins, "An Appraisal of Program Specifications," MIT Project MAC, Computation Structures Group Memo 141-1, April 1977.

[Lloyd84] J.W. Lloyd, "Foundations of Logic Programming," Springer-Verlag, 1984.

[McMullin81] P.R. McMullin, J.D. Gannon, "Evaluating a Data Abstraction Testing System Based on Formal Specifications," Journal of Systems and Software, vol. 2, pp. 177-186, 1981.

[McMullin82] P.R. McMullin, J.D. Gannon, M.D. Weiser, "Implementing a Compiler-Based Test Tool," Software Practice and Experience, vol. 12, pp. 971-979, 1982.

[McMullin83] P.R. McMullin, J.D. Gannon, "Combining Testing with Formal Specifications: A Case Study," IEEE TSE, vol. 9, no. 3, pp. 328-334, May, 1983.

[Meyers74] G.J. Meyers, C. Heuerman, J. Winterton, "Automated Test and Verification," IBM Tech. Bulletin, vol. 17, no. 7, pp. 2030-2035, 1974.

[Meyers76] G.J. Meyers, "Software Reliability: Principles and Practices," pp. 169-215, John Wiley and Sons, 1976.

[Moitra79] A. Moitra, "Direct Implementation of Algebraic Specification of Abstract Data Types," Tech. Rep. 48., NCSDCT, TIFR, Bombay, 1979.

[Pesch85] H. Pesch, P. Schnupp, H. Schaller, A.S. Spirk, "Test Case Generation Using Prolog," Proc. 8th Int. Conf. on Software Engineering, pp. 252-258, Aug., 1985.

[Richardson81] D.J. Richardson, L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," PROC. 5th. Int. Conf. on Software Engineering, pp. 244-253, March 1981.

[Richardson85] D.J. Richardson, L.A. Clarke, "Partition Analysis: A Method Combining Testing and Verification," IEEE TSE, vol. 11, no. 12, pp. 1477-1490, Dec., 1985.

[Richardson88] D.J. Richardson, M.C. Thompson, "The RELAY Model of Error Detection and its Application," PROC. 2nd. Workshop on Software Testing, Verification and Analysis, pp. 223-230, July 1988.

[Stanat77] D.F. Stanat, D.F. McAllister, "Discrete Mathematics in Computer Science," Prentice Hall, 1977.

[Tennent81] R.D. Tennent, "Principles of Programming Languages," Prentice Hall, 1981.

[Thatcher82] J.W. Thatcher, E.G. Wagner, J.B. Wright, "Data Type Specification: Parameterization and the Power of Specification Techniques," ACM TOPLAS, vol. 4, no. 4, pp. 711-732, Oct., 1982.

[Thomas88] P. Thomas, H. Robinson, J. Emms, "Abstract Data Types: Their Specification, Representation, and Use," pp. 189-202, Clarendon Press, 1988.

[Travendale85] R.D. Travendale, "A Technique for Prototyping Directly from a Specification," Proc. 8th. Intl. Conf. on Software Engineering, pp. 224-229, Aug., 1985.

[Urban82] J.E. Urban, "Software Development With Executable Functional Specifications," Proc. 6th. Intl. Conf. on Software Engineering, pp. 418-419, Sept., 1982.

[Weber86] H. Weber, H. Ehrig, "Specification of Modular Systems," IEEE TSE, vol. 12, no. 7, pp. 784-798, July 1986.

[Weyuker80] E.J. Weyuker, T.J. Ostrand "Theories of Program Testing and the Application of Revealing Subdomains," IEEE TSE, vol. 6, no. 3, May 1980, pp. 236-246.

[Weyuker82] E.J. Weyuker, "On Testing Non-testable Programs," The Computer Journal, vol. 25, no.4, pp. 465-470, 1982.

[White80] L.J. White, "A Domain Strategy for Computer Program Testing," IEEE TSE, vol. 6, no. 3, pp. 247-257, May 1980.

[White87] L.J. White, "Software Testing and Verification," in: Advances in Computers, vol. 26, pp. 337-391, 1987.

[Wild86] C. Wild, D. Eckhardt, A. Pang, S. Sundararajan, "Analysis of Executable Specifications for Testing and Monitoring Abstract data Types," Dept. of Computer Science, Old Dominion University, Norfolk, VA, March, 1986.

[Zave84] P. Zave, "The Operational Versus the Conventional Approach to Software Development," CACM, vol. 27, no. 2, pp. 104-118, Feb., 1984.

Appendix I: An Algebraic Specification for Type List

The following is an algebraic specification of our sample abstract data type "List". The hidden functions, as discussed in section 3.1.7, are "move_previous," "atend," and "is_single."

Type list

SYNTAX

initList()	->list
addElt(list,integer)	->list
deleteElt(list)	->list
next(list)	->list
previous(list)	->list
setElt(list,integer)	->list
emptyList(list)	->boolean
getElt(list)	->integer
size(list)	->integer
includes(list,integer)	->boolean
removeDups(list)	->list

SEMANTICS

Declare l1:list,l2:list,i1:integer,i2:integer

1) getElt(initList())	=NULL
2) getElt(addElt(l1,i1))	=i1
3) getElt(previous(l1))	=IF emptyList(l1) THEN initList() ELSE getElt(move_previous(l1))
4) next(initList())	=initList()
5) next(previous(l1))	=l1
6) next(addElt(l1,i1))	=IF atend(l1) THEN addElt(l1,i1)

```
                                ELSE addElt(setElt(next(l1),i1),getElt(next(l1)))

7)  deleteElt(initList())       =initList()
8)  deleteElt(addElt(l1,i1))    =l1
9)  deleteElt(previous(l1))     =IF emptyList(l1) THEN previous(l1)
                                 ELSE previous(addElt(deleteElt(deleteElt(l1)),getElt(l1)))

10) setElt(initList(),i1)       =addElt(initList(),i1)
11) setElt(addElt(l1,i1),i2)    =addElt(l1,i2)
12) setElt(previous(l1),i1)     =addElt(deleteElt(l1),i1)

13) emptyList(initList())           =true
14) emptyList(addElt(l1,i1))        =false
15) emptyList(previous(l1))         =emptyList(l1)

16) size(initList())                =0
17) size(addElt(l1,i1))             =size(l1)+1
18) size(previous(l1))              =size(l1)

19) includes(initList(),i1)         =false
20) includes(addElt(l1,i1),i2)      =IF  I1=I2  THEN true
                                     ELSE includes(l1,i2)
21) includes(previous(l1),i1)       =includes(l1,i1)

22) removeDups(initList())          =initList
23) removeDups(addElt(l1,i1))       =IF includes(l1,i1) THEN removeDups(l1)
                                     ELSE addElt(removeDups(l1),i1))
24) removeDups(previous(l1))        =removeDups(l1)

25) atend(initList())               =true
26) atend(previous(l1))             =IF emptyList(l1) OR is_single(l1) THEN true
                                     ELSE false
27) atend(addElt(l1,i1))            =atend(l1).

28) is_single(previous(l1))         =is_single(l1)
29) is_single(initList())           =false
30) is_single(addElt(l1,i1))        =emptyList(l1)

31) move_previous(initList())       =initList()
32) move_previous(previous(l1))     =IF emptyList(move_previous(l1)) THEN l1
```

```
                                    ELSE move_previous(move_previous(l1)
33) move_previous(addElt(l1,i1))    =IF emptyList(l1) THEN addElt(l1,i1)
                                    ELSE l1

END.
```

Appendix II: T-3

In this appendix we will present a detailed outline of our Type Testing Tool (T-3). This includes listings of the main shell scripts and significant routines. We also include samples of important files that are passed between routines. At the end of this appendix we demonstrate how T-3 works for both correct and incorrect MODULA-2 implementations of a simple queue. In section 3.3 we outlined our new software testing methodology that T-3 implements. In section 3.4 we described the general operation of T-3.

Input Files

As outlined in sections 3.3 and 3.4, T-3 needs only the specification and implementation of an ADT. The implementation of an ADT in MODULA-2 actually consists of two files. The "name.mod" file contains the "code" and the "name.def" file contains "header" information that allows independent compilation. The ADT specification consists of the file "alg_spec". For this version of T-3 we also include a file called "name.functions". This is a short file that lists the various functions in the ADT. This file is produced separately to allow separation of instance class production from the rest of the system. It could be derived from "alg.spec" if that were required. Listing A2.1 is a listing of the *name.mod* file for a simple queue type. Listing A2.2 is a listing of the corresponding *name.def* file. Listing A2.3 is a listing of the

corresponding *alg_spec* file, and listing A2.4 is a listing of the corresponding name.functions file.

```
IMPLEMENTATION MODULE Queue;

FROM Storage IMPORT
        ALLOCATE, DEALLOCATE;

TYPE    queue = POINTER TO queuedata;
        queuedata =
                RECORD
                 val : INTEGER;
                 next: queue
                END;


PROCEDURE newq():queue;
        VAR q : queue;
BEGIN
        NEW(q);
        q^.val := -1;
        q^.next := NIL;
        RETURN q ;
END newq;



PROCEDURE add (q:queue; i:INTEGER): queue;
        VAR r,s: queue;
BEGIN
        NEW(r);
        r^.val := i ;
        r^.next := q;
        s := q^.next;
        WHILE s <> NIL DO
                r^.next := s;
                s := s^.next
        END;
        RETURN q;
END add;
```

```
PROCEDURE remove (q:queue) : queue;
        VAR r:queue;
BEGIN
        IF q^.next = NIL
          THEN RETURN q
          ELSE
                r := q^.next;
                DISPOSE (q);
                RETURN r
        END;
END remove;


PROCEDURE front (q:queue): INTEGER;
BEGIN
        RETURN (q^.val);
END front;


PROCEDURE isnewq (q:queue) : BOOLEAN;
BEGIN
        RETURN ((q^.next) = NIL);
END isnewq;

END Queue.
```

Listing A2.1: Queue.mod

```
DEFINITION MODULE Queue;

EXPORT QUALIFIED
        queue,
        newq, add, remove, front, isnewq;

TYPE queue;

PROCEDURE newq() :queue;

PROCEDURE add (q:queue; i:INTEGER): queue;

PROCEDURE remove (q:queue) : queue;

PROCEDURE front (q:queue): INTEGER;
```

```
PROCEDURE isnewq (q:queue) : BOOLEAN;

END Queue.
```

Listing A2.2: Queue.def

```
Type queue

SYNTAX
        newq()                  ->queue
        add(queue,integer)  ->queue
        remove(queue)           ->queue
        front(queue)            ->integer
        isnewq(queue)           ->boolean

SEMANTICS

        Declare q:queue,i:integer

1) isnewq(newq())       =true
2) isnewq(add(q,i))     =false
3) remove(newq())       =newq
4) remove(add(q,i))     =IF isnewq(q) THEN newq()
                                ELSE add(remove(q),i)
5) front(newq)                  =-1
6) front(add(q,i))      =IF isnewq(q) THEN i
                                ELSE front(q)

END.
```

Listing A2.3: Queue.spec

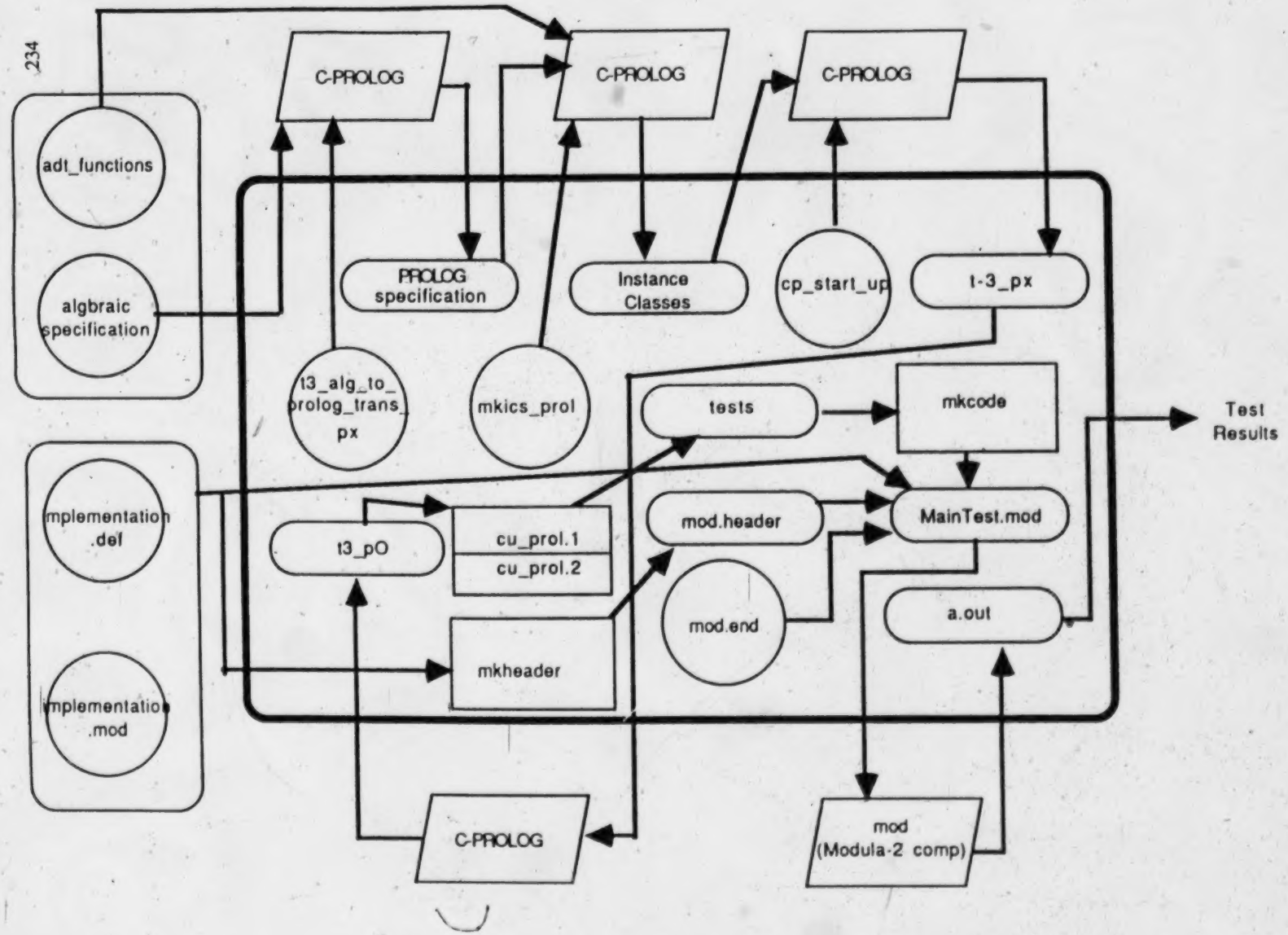
```
i_type(newq).
e_type(remove(adt)).
c_type(add(adt,v)).
o_type(front(adt,ans)).
o_type(isnewq(adt,ans)).
```

Listing A2.4: queue_functions

Operation

The operation of the overall T-3 system is shown in figure A2.1.

Figure A2.1: Operation of T-3



Given the four input files (two for implementation, two for specification) outlined in the previous section, T-3 is invoked by giving the following two commands:

instance_classes *name.functions* >*ic.file*

T-3 *alg_spec name ic.file*

"T-3" and "instance_classes" are simple shell scripts and are listed in Listings A2.5 and A2.6.

```
echo "[" > /tmp/$$_x
echo $1 |cat /tmp/$$_x - >/tmp/$$_xx
echo ",'/u1/grad/don/T-3/ics/mkics_prol'],do_to_level(6)." | cat /tmp/$$_xx - > /tmp/$$_xxx
cat  /tmp/$$_xxx |sicstus
rm /tmp/$$_x /tmp/$$_xx /tmp/$$_xxx
```

Listing A2.5: "instance_classes" Script.

```
echo "T-3 OPUS-3"
echo
/usr/cavell/u1/grad/don/T-3/OPUS-3 $1 $2 $3
```

Listing A2.6: "T-3" Script.

The operational part of "instance_classes" command is */u1/grad/don/T-3/ics/mkics_prol*. Listing A2.7 gives the PROLOG code for mkics_prol.

```
first_line_variable(1).
```

```
second_line_variable(1).


build_first_line_level(0):-!.

build_first_line_level(1):- i_type(F),atomic(F), name(F,L1),
                    concatenate(L1,[40,41],L2), name(Fo,L2),
                    asserta(first_line_level(1,Fo)),fail.

build_first_line_level(1):- i_type(F), + (atomic(F)), replace_vs(F,Fo),
                    asserta(first_line_level(1,Fo)),fail.

build_first_line_level(1):- !.

build_first_line_level(N):- M is N - 1, (c_type(F);e_type(F)),
                    first_line_level(M,F1),
                    replace_vs(F,Fx), replace_adt(Fx,F1,Fo),
                    asserta(first_line_level(N,Fo)), fail.

build_first_line_level(N):- !.



build_second_line_level(0):- !.

build_second_line_level(1):- i_type(F),atomic(F),asserta(second_line_level(1,F)),fail.

build_second_line_level(1):- i_type(F), + (atomic(F)), replace_vs_2(F,Fo),
                    asserta(second_line_level(1,Fo)), fail.

build_second_line_level(1):- !.

build_second_line_level(N):- M is N-1, (c_type(F);e_type(F)),
                    second_line_level(M,F1),
                    replace_vs_2(F,Fx),replace_adt(Fx,F1,Fo),
                    asserta(second_line_level(N,Fo)),
                    fail.

build_second_line_level(N):- !.



build_output_first_line(Level):- first_line_level(Level,F),
                    o_type(O),replace_ans(O,Ox), replace_adt(Ox,F,Fo),
                    asserta(first_output_line(Level,Fo)), fail.

build_output_first_line(Level):- !.

build_output_second_line(Level):- second_line_level(Level,F),
          o_type(O),replace_ans(O,Ox), replace_adt(Ox,F,Fo),
          asserta(second_output_line(Level,Fo)), fail.
```

```
build_output_second_line(Level):- !.


write_level(N):- get_output_lines(N,F1,F2),
                 format(""nwrite(''***w.''),~n~w.~n",[F1,F2]), fail.

write_level(N):- !.


get_output_lines(Level,Line1,Line2):- retract(first_output_line(Level,Line1)),
                 retract(second_output_line(Level,Line2)).


do_to_level(0):- !.
do_to_level(N):- M is N-1, do_to_level(M),
                 build_first_line_level(N),
                 build_second_line_level(N),
                 build_output_first_line(N),
                 build_output_second_line(N),
                 write_level(N).


replace_ans(Fi,Fo):- Fi =.. Li, replace(ans,'Answer',Li,Lo), Fo =.. Lo, !.

replace_adt(Fi,A,Fo):- Fi =.. Li, replace(adt,A,Li,Lo), Fo =.. Lo, !.


replace_vs(Fi,Fo):- Fi =.. Li, member(v,Li), retract(first_line_variable(N)),
                 N1 is N + 1, asserta(first_line_variable(N1)),
                 name(N,Ln), concatenate([86],Ln,Lv),name(V,Lv),
                 replace(v,V,Li,Lo), Fx =.. Lo, !,
                 replace_vs(Fx,Fo).

replace_vs(Fo,Fo):- !.


replace_vs_2(Fi,Fo):- Fi =.. Li, member(v,Li), retract(second_line_variable(N)),
        N1 is N + 1, asserta(second_line_variable(N1)),
        name(N,Ln), concatenate([86],Ln,Lv),name(V,Lv),
        replace(v,V,Li,Lo), Fx =.. Lo, !,
        replace_vs_2(Fx,Fo).

replace_vs_2(Fo,Fo):- !.


replace(I,O,[I|L],[O|L]):- !.
```

```
replace(I,O,[X|Li],[X|Lo]):- replace(I,O,Li,Lo).

concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]):- concatenate(L1,L2,L3).

member(X,[X|_]).
member(X,[_|L]):- member(X,L).
```

Listing A2.7: Code For Simple Instance Class Generation.

The operational part of the "T-3" command is OPUS-3. OPUS-3 is also a shell script and is given in Listing A2.8

```
#       This command should be invoked as follows:
#
#               OPUS-3  alg_spec imp ic
#
#
#       WHERE:
#               alg_spec is the file containing the algebraic
#                               specification of the ADT.
#
#               imp.mod         is the file containing the MODULA-2
#                               implementation of the ADT.
#                               NOTE: This will actually be 2 files:
#                                       imp.mod
#                                       imp.def
#
#
#               ic              is the file containing the instance
#                               classes.
#                               <FORMAT of t3_tmp_pl>
#
#.....................................................................

set pa1="/u1/grad/don/T-3/OPUS-3.files"
set adt="$2"

set path =($pa1 . ~/bin  /usr/ucb /bin /usr/new/bin /usr/bin /usr/local/bin )
#.....................................................................

mkdir  /tmp/$$

cp $3 /tmp/$$/t3_pl                                     #produce  instance classes
```

```
cp $1 /tmp/$$/sys_alg_spec                          # copy algebraic spec over

cp $adt.mod  /tmp/$$/$adt.mod                              #produce ADT  implementation with (coverage meas
                                                           #monitoring  code.

cp $adt.def  /tmp/$$/$adt.def                       #copy definition  module  over .

#.....


cd /tmp/$$
cp $pa1/StdMonitoring.* .
mod StdMonitoring.mod >& m_garb
mod adt.mod >& m_garb

echo $adt|cat - sys_alg_spec|mkheader>mod.header    #make  header  of  MainTest.mod file

cat $pa1/determine_constructors |cprolog $pa1/t3_alg_to_prolog_trans_px >cp_garb
                                                           #determine constructor functions
cat $pa1/produce_prolog_specs |cprolog $pa1/t3_alg_to_prolog_trans_px >cp_garb
                                                           #make PROLOG version of  specifications

cat $pa1/cp_start_up | cprolog>& cp_garb            #Start PROLOG process with specs

cat t3_pl | cprolog t3_px>& t3_pO                   #Feed ic's  to  PROLOG process

cu_prol.1 < t3_pO |cu_prol.2 > tests                #clean up PROLOG  output

mkcode <tests|cat mod.header - $pa1/mod.end>MainTest.mod   #make test prog

mod  $adt.mod StdMonitoring.mod MainTest.mod               #Compile  this  test

a.out                                                      #run this  test

cd ..
rm -r /tmp/$$
```

Listing A2.8: OPUS-3 Script.

Sample Runs

The implementation module in listing A2.1 is an actual implementation the author produced while implementing the queue type which contains an error. Below are three runs of T-3. The first is for the implementation given in listing A2.1. The second is for an implementation with a comma error. The third is for what we believe is a correct implementation.

```
Script started on Fri Jan 19 11:24:41 1990
% pwd
/u1/grad/don/private/thesis.work/tests/OPUS-3
% ls
Error1.mod      Error3.mod      Queue.mod       correct.def     holding.mod     t3_tmp_pI.2
Error2.mod      Queue.def       Queue.spec      correct.mod     t3_tmp_pI       typescript
% cp Error1.mod Queue.mod                <copy "Error.1" into implementation file>
% T-3 Queue.spec Queue t3_tmp_pI.2       <Run T-3 with "Error.1">
T-3  OPUS-3


Implementation Failed  on this  pass  on test #1

% cp Error2.mod Queue.mod                <copy "Error.2" into implementation file>
% T-3 Queue.spec Queue t3_tmp_pI.2       <Run T-3 with "Error.2">
T-3  OPUS-3

File Queue.mod, line 29: syntax error
        VAR r,s,: queue;

File Queue.mod, line 31: r: Symbol not found
File Queue.mod, line 32: r: Symbol not found
File Queue.mod, line 33: r: Symbol not found
File Queue.mod, line 34: s: Symbol not found
File Queue.mod, line 35: s: Symbol not found
File Queue.mod, line 36: s: Symbol not found
File Queue.mod, line 36: r: Symbol not found
File Queue.mod, line 37: s: Symbol not found
File Queue.mod, line 37: s: Symbol not found
File Queue.mod, line 41: r: Symbol not found
File Queue.mod, line 41: Return value not assignable to function result
12 parsing errors
a.out: Permission denied.

% cp correct.mod Queue.mod              <copy "correct" into implementation file>
```

```
% T-3 Queue.spec Queue t3_tmp_pl.2          <Run T-3 with a "correct" implementation>
T-3  OPUS-3

All tests completed successfully on this pass.

%
script done on Fri Jan 19 11:39:10 1990
```

Listing A2.9: Sample Runs of T-3

Appendix III: A Large Example

In this appendix we give the details of producing the test set for a large example ADT. Figure A3.1 gives the algebraic specification for an extended list type called "List-e". This ADT was originally given as a MODULA-2 assignment to a third year undergraduate class at the University of Alberta.

Type list-e

SYNTAX

initList()	->list-e
addElt(list-e,integer)	->list-e
deleteElt(list-e)	->list-e
next(list-e)	->list-e
previous(list-e)	->list-e
setElt(list-e,integer)	->list-e
emptyList(list-e)	->boolean
getElt(list-e)	->integer
size(list-e)	->integer
intersect(list-e,list-e)	->list-e
union(list-e,list-e)	->list-e
includes(list-e,integer)	->boolean
removeDups(list)	->list

SEMANTICS

Declare l1:list,l2:list-e,i1:integer,i2:integer

1) getElt(initList()) =NULL
2) getElt(addElt(l1,i1)) =i1

```
3)  getElt(previous(l1))        =IF emptyList(l1) THEN NULL
                                 ELSE getElt(move_previous(l1))

4)  next(initList())            =initList()
5)  next(previous(l1))          =l1
6)  next(addElt(l1,i1))         =IF atend(l1) THEN addElt(l1,i1)
                                 ELSE addElt(setElt(next(l1),i1),getElt(next(l1)))

7)  deleteElt(initList())       =initList()
8)  deleteElt(addElt(l1,i1))    =l1
9)  deleteElt(previous(l1))     =IF emptyList(l1) THEN previous(l1)
                                 ELSE previous(addElt(deleteElt(deleteElt(l1)),getElt(l1)))

10) setElt(initList(),i1)       =addElt(initList(),i1)
11) setElt(addElt(l1,i1),i2)    =addElt(l1,i2)
12) setElt(previous(l1),i1)     =addElt(deleteElt(previous(l1)),i1)

13) emptyList(initList())           =true
14) emptyList(addElt(l1,i1))        =false
15) emptyList(previous(l1))         =emptyList(l1)

16) size(initList())                =0
17) size(addElt(l1,i1))             =size(l1)+1
18) size(previous(l1))              =size(l1)

19) includes(initList(),i1)         =false
20) includes(addElt(l1,i1),i2)      =IF  I1=I2  THEN true
                                     ELSE includes(l1,i2)
21) includes(previous(l1),i1)       =includes(l1,i1)

22) intersect(initList(),l2)        =initList
23) intersect(addElt(l1,i1),l2)     =IF includes(l2,i1) AND
                                    NOT (includes(intersect(l1,l2),i1))
                                    THEN addElt(intersect(l1,l2),I)
                                    ELSE intersect(l1,l2)
24) intersect(previous(l1),l2)      =intersect(l1,l2)

25) union(initList(),l2)            =removeDups(l2)
26) union(addElt(l1,i1),l2)         =IF includes(union(l1,l2),i1)
                                     THEN union(l1,l2)
```

```
                                    ELSE addElt(union(l1,l2),i1)
27) union(previous(l1),l2)          =union(l1,l2)

28) removeDups(initList())          =initList
29) removeDups(addElt(l1,i1))       =IF includes(l1,i1) THEN removeDups(l1)
                                    ELSE addElt(removeDups(l1),i1))
30) removeDups(previous(l1))        =removeDups(l1)

31) atend(initList())               =true
32) atend(previous(l1))             =IF emptyList(l1) OR is_single(l1) THEN true
                                    ELSE false
33) atend(addElt(l1,i1))            =atend(l1).

34) is_single(previous(l1))         =is_single(l1)
35) is_single(initList())           =false
36) is_single(addElt(l1,i1))        =emptyList(l1)

37) move_previous(initList())       =initList()
38) move_previous(previous(l1))     =IF emptyList(move_previous(l1)) THEN l1
                                    ELSE move_previous(move_previous(l1))
39) move_previous(addElt(l1,i1))    =IF emptyList(l1) THEN addElt(l1,i1)
                                    ELSE l1

END.
```

Figure A3.1: Algebraic Specification for a Large Example

Generation

To generate our test set we started the SICStus PROLOG complier and loaded the necessary PROLOG files as follows:

```
:- [list_prolog_spec,list_prolog_startup].
:- [build_level,work_level_cost_calc,utilities].
:- [work_level_benefit_calc_2,assume_tested,upper_level].
:- [work_canonical_form].
```

The key files here are "list_prolog_spec", "list_prolog_startup", "upper_level", and "work_level_benefit_calc_2". These files are listed in listings A3.1 through A3.4 respectively.

```
getElt(initList,null,CI,CI).
getElt(addElt(L1,I1),I1,CI,CI).
getElt(previous(L1),null,CI,CO):-       emptyList(L1,true,CI,CO).
getElt(previous(L1),getElt(X),CI,CO):-  emptyList(L1,false,CI,CA),move_previous(L1,X,CA,CO).

next(initList,initList,CI,CI).
next(previous(L1),L1,CI,CI).
next(addElt(L1,I1),addElt(L1,I1),CI,CO):- atend(L1,true,CI,CO).
next(addElt(L1,I1),addElt(X,Y),CI,CO):- atend(L1,false,CI,CA),setElt(next(L1),I1,X,CA,CB),getElt(next(L1),Y,CB,CO).

deleteElt(initList,initList,CI,CI).
deleteElt(addElt(L1,I1),L1,CI,CI).
deleteElt(previous(L1),previous(L1),CI,CO):-  emptyList(L1,true,CI,CO) .
deleteElt(previous(L1),previous(addElt(X,Y)),CI,CO):-     emptyList(L1,false,CI,CA), deleteElt(L1,Z,CA,CB),deleteElt(Z,X,CB,C(


setElt(initList,I1,addElt(initList,I1),CI,CI).
setElt(addElt(L1,I1),I2,addElt(L1,I2),CI,CI).
setElt(previous(L1),I1,addElt(X,I1),CI,CO):-      deleteElt(previous(L1),X,CI,CO).

emptyList(initList,true,CI,CI).
```

```
emptyList(addElt(L1,I1),false,CI,CI).
emptyList(previous(L1),X,CI,CO):- emptyList(L1,X,CI,CO).

size(initList,0,CI,CI).
size(addElt(L1,I1),X,CI,CO):- size(L1,Y,CI,CO), X is Y+1.
size(previous(L1),X,CI,CO):-          size(L1,X,CI,CO).

includes(initList,I1,false,CI,CI).
includes(addElt(L1,I1),I2,true,CI,CO):- constraint(I1=I2,CI,CO).
includes(addElt(L1,I1),I2,X,CI,CO):- constraint(I1 == I2,CI,CA),includes(L1,I2,X,CA,CO).
includes(previous(L1),I1,X,CI,CO):- includes(L1,I1,X,CI,CO).

intersect(initList,L2,initList,CI,CI).
intersect(addElt(L1,I1),L2,addElt(X,I1),CI,CO):- includes(L2,I1,true,CI,CA),intersect(L1,L2,X,CA,CB),includes(X,I1,false,CB,CO)
intersect(addElt(L1,I1),L2,X,CI,CO):- includes(L2,I1,false,CI,CA),intersect(L1,L2,X,CA,CO).
intersect(addElt(L1,I1),L2,X,CI,CO):- includes(L2,I1,true,CI,CA),intersect(L1,L2,X,CA,CB),includes(X,I1,true,CB,CO).
intersect(previous(L1),L2,X,CI,CO):- intersect(L1,L2,X,CI,CO).

union(initList,L2,X,CI,CO):- removeDups(L2,X,CI,CO).
union(addElt(L1,I1),L2,X,CI,CO):- union(L1,L2,X,CI,CA),includes(X,I1,true,CA,CO).
union(addElt(L1,I1),L2,addElt(X,I1),CI,CO):- union(L1,L2,X,CI,CA),includes(X,I1,false,CA,CO).
union(previous(L1),L2,X,CI,CO):- union(L1,L2,X,CI,CO).

removeDups(initList,initList,CI,CI).
removeDups(addElt(L1,I1),X,CI,CO):- includes(L1,I1,true,CI,CA), removeDups(L1,X,CA,CO).
removeDups(addElt(L1,I1),addElt(X,I1),CI,CO):- includes(L1,I1,false,CI,CA), removeDups(L1,X,CA,CO).
removeDups(previous(L1),X,CI,CO):- removeDups(L1,X,CI,CO).

atend(initList,true,CI,CI).
atend(previous(L1),true,CI,CO):- emptyList(L1,true,CI,CA); is_single(L1,true,CA,CO).
atend(previous(L1),false,CI,CO):-emptyList(L1,false,CI,CA), is_single(L1,false,CA,CO).
atend(addElt(L1,I1),X,CI,CO):- atend(L1,X,CI,CO).

is_single(previous(L1),X,CI,CO):- is_single(L1,X,CI,CO).
is_single(initList,false,CI,CI).
is_single(addElt(L1,I1),X,CI,CO):- emptyList(L1,X,CI,CO).

move_previous(initList,initList,CI,CI).
move_previous(previous(L1),L1,CI,CO):- emptyList(L1,true,CI,CO).
move_previous(previous(L1),X,CI,CO):- emptyList(L1,false,CI,CA),move_previous(L1,Y,CA,CB),move_previous(Y,X,CB,CO).
move_previous(addElt(L1,I1),addElt(L1,I1),CI,CO):-  emptyList(L1,true,CI,CO).
move_previous(addElt(L1,I1),L1,CI,CO):- emptyList(L1,false,CI,CO).
```

Listing A3.1: list_prolog_spec

```
level(0,initList,4,1,-1).
best_b_c(0,initList,0.25).

tested_trace(n_o_n_e).
tested_arc(n_o_n_e).

deepest_level(0).

variable(i1).
variable(toi).

o_function(includes(toi,i1)).
o_function(emptyList(toi)).
o_function(getElt(toi)).
o_function(size(toi)).

i_function(initList).

c_function(addElt).
c_function(previous).

internal_function(atend).
internal_function(is_single).
internal_function(move_previous).

toi_function(addElt(toi,i1)).
toi_function(deleteElt(toi)).
toi_function(next(toi)).
toi_function(previous(toi)).
toi_function(setElt(toi,i1)).
toi_function(intersect(toi,toi)).
toi_function(union(toi,toi)).
toi_function(removeDups(toi)).

external_function(initList).
external_function(addElt).
external_function(deleteElt).
external_function(next).
external_function(previous).
external_function(setElt).
external_function(emptyList).
external_function(getElt).
external_function(size).
external_function(intersect).
external_function(union).
external_function(includes).
external_function(removeDups).
```

Listing A3.2: list_prolog_startup

```
do_level(L):- build_level(L), level_cost_calc(L), level_benefit_calc(L).


accept_best_level(L):- best_b_c(L,T,BC),
                       display(T),
                       display('     is being marked as TESTED.'),
                       nl,display('BC= '),
                       display(BC),nl,
                       display('Level= '),
                       display(L),nl,
                       level(L,T,C,B,X),
                       display('Cost=  '),
                       display(C),nl,
                       display('Benefit=  '),
                       display(B),nl,
                       assume_tested(T).


accept_best_all:- find_best_all(1,L), accept_best_level(L).


take_best:- accept_best_all,re_benefit_all.
take_first_20:- take_best,take_best,take_best,take_best,take_best,
                take_best,take_best,take_best,take_best,take_best,
                take_best,take_best,take_best,take_best,take_best,
        take_best,take_best,take_best,take_best,take_best.

go:- take_best,go.


find_best_all(Cur_lev,L):- deepest_level(Cur_lev),!.
find_best_all(Cur_lev,L):- Next is Cur_lev + 1, find_best_all(Next,L),
                       best_b_c(L,T,BC), best_b_c(Cur_lev,Tc,BCc),
                       BC > BCc, !.
find_best_all(L,L).


re_benefit_all:- deepest_level(End), re_benefit_to_level(0,End).
re_benefit_to_level(L,L):- level_benefit_calc(L),!.
re_benefit_to_level(L,E):- level_benefit_calc(L), L1 is L+1,
                       re_benefit_to_level(L1,E).


print_level(L,File):- tell(File),pr_l(L),nl,told.
```

```
pr_l(L):- level(L,T,C,B,BC), write(level(L,T,C,B,BC)), nl, fail.
pr_l(L).


print_tree(File):- tell(File),multi_level_print(1), nl, told.


multi_level_print(L):- deepest_level(L),!, nl, nl, write('*****  Level '),
                       write(L),write('  *****'), nl, nl,
                       pr_l(L).
multi_level_print(L):- nl, nl, write('*****  Level '),
                       write(L),write('  *****'), nl, nl,
                       pr_l(L), Next is L + 1,
                       multi_level_print(Next).
```

Listing A3.3: upper_level

```
:- [work_trace_benefit_2].

level_benefit_calc(L):- assertz(level(L,-1,-1,-1,-1)),
                        retract(best_b_c(L,T,BC)),
                        asserta(best_b_c(L,0,-1)),fail.
level_benefit_calc(L):- retract(level(L,T,C,Bi,BCi)), T == -1,
                        trace_benefit_2(T,B), BC is B/C,
                        check_best_b_c(L,T,BC),
                        assertz(level(L,T,C,B,BC)),
                        fail.
level_benefit_calc(L).


check_best_b_c(L,T,BC):- best_b_c(L,To,BCo), BC>BCo,
                        retract(best_b_c(L,To,BCo)), asserta(best_b_c(L,T,BC)),!.
check_best_b_c(L,T,BC).


sub_traces([],[],[]):- !.
sub_traces(I,[],[]):- atomic(I),(variable(I); internal_function(I)),!.
sub_traces(I,[[I]],[]):- atomic(I),!.
sub_traces(I,Nt,Ot):- I=..[F|Ops],(variable(F); internal_function(F)),!,
                        Ops=[Ops1|Opsx],sub_traces(Ops1,Nt,Ot).
sub_traces(I,Nt,Ot):- I=..[F|Ops],Ops=[Ops1|Opsx],sub_traces(Ops1,Nr,Or),
                        append(Nr,Or,Ot),
                        build_traces(F,Nr,Nt).


build_traces(F,[],[[F]]):- !.
build_traces(F,[T|Rest],Out):- append([F],T,Tn),
                        build_traces(F,Rest,Tr),append([Tn],Tr,Out).


remove_bad_traces([],[]):- !.
remove_bad_traces([A|B],C):- (member(A,B);tested_trace(A)),!,
                        remove_bad_traces(B,C).
remove_bad_traces([A|B],[A|C]):- remove_bad_traces(B,C).
```

Listing A3.4: work_level_benefit_calc_2

Results

Using the PROLOG programs listed above we used the methodology given in chapter 8 to produce the following series of instance classes to be tested. These represent the first 25 test cases to be run. For clarity we have also listed the test cases associated with each instance class. It took a SICStus PROLOG interpreter 48 hours on a MIPS M1000 and a system load below 2.5 to produce these test cases. The reason this ADT took longer that the others is the functions "intersect" and "union" accept two TOI inputs which leads to a rapid growth of the number of sub-instance classes to be examined. This problem is discussed in section 8.4. while this example took a significant amount of computer time, we note that previous methodologies simply could not handle this example at all.

As we stated in section 8.4, the results of this example are very similar to the results of the example in figures 8.6 and 8.7 and Tables 8.2 and 8.3. The discussion of these results is given in sections 8.3 and 8.4. We could not run a validation experiment for this example, as we described in section 8.5, to compare our methodology to previous ones [Bouge85b, Choquet86, Wild86, Bouge86] because those previous methodologies do not work on this example. The reasons previous methodologies do not work on this example are discussed in section 8.4.

removeDups(deleteElt(removeDups(initList)))
TESTS:
emptyList(removeDups(deleteElt(removeDups(initList))))
getElt(removeDups(deleteElt(removeDups(initList))))
size(removeDups(deleteElt(removeDups(initList))))
includes(removeDups(deleteElt(removeDups(initList))),I1)

removeDups(addElt(previous(initList)),I1)
TESTS:
emptyList(removeDups(addElt(previous(initList)),I1))
getElt(removeDups(addElt(previous(initList)),I1))
size(removeDups(addElt(previous(initList)),I1))
includes(removeDups(addElt(previous(initList)),I1),I2); I1 ≠ I2
includes(removeDups(addElt(previous(initList)),I1),I2); I1 = I2

deleteElt(removeDups(previous(initList)))
TESTS:
emptyList(deleteElt(removeDups(previous(initList))))
getElt(deleteElt(removeDups(previous(initList))))
size(deleteElt(removeDups(previous(initList))))
includes(deleteElt(removeDups(previous(initList))),I1)

removeDups(removeDups(deleteElt(removeDups(addElt(initList,I1)))))

TESTS:

emptyList(removeDups(removeDups(deleteElt(removeDups(addElt(initList,I1))))
getElt(removeDups(removeDups(deleteElt(removeDups(addElt(initList,I1))))))
size(removeDups(removeDups(deleteElt(removeDups(addElt(initList,I1))))))
includes(removeDups(removeDups(deleteElt(removeDups(addElt(initList,I1))))),

addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4)

TESTS:

emptyList(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4))
getElt(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4))
size(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4))
includes(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4),I5); I1 = I5
includes(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4),I5); I2 = I5
includes(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4),I5); I3 = I5
includes(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4),I5); I4 = I5
includes(addElt(addElt(addElt(addElt(initList,I1),I2),I3),I4),I5); I1 ≠ I5 AND
I2 ≠ I5 AND I3 ≠ I5 AND I4 ≠ I5

Appendix IV: A Detailed PFS Calculation

In section 8.3 we gave an example of our test case generation method for the type Bag-c. Table 8.2 gave the first thirty tests generated for that type. For convenience, table 8.2 is reproduced below. In this appendix we list exactly which elements are removed from the possible failure set (PFS) for that example.

First 30 Tests for Search to Depth 6					
Trace Ordering	Trace Sentence	Level	# of Test Cases	PFS Elements Removed	B/C
1	E,D,I	3	3	5	5/9=0.56
2	A,D,I	3	4	4	4/12=0.33
3	E,E,D,D,E,I	6	3	5	5/18=0.28
4	D,E,D,E,E,I	6	3	4	4/18=0.22
5	E,D,E,D,D,I	6	3	4	4/18=0.22
6	D,D,D,E,A,I	6	7	9	9/42=0.21
7	A,A,A,A,I	5	7	6	6/35=0.17
Total			30	37	

NOTE: E= removeElt
D= removeDups
I= initBag
A= addElt

Table 8.2: Order of Traces Tested for Type Bag-c

The reasons why particular elements are removed from the PFS are discussed in section 8.2. The algorithm we used to determine how many PFS elements are removed by testing an instance class is given in Figure A4.1.

For the first trace tested there is only one canonical form. Therefore we have tested EDI as a trace to the equivalence space of bags with no elements. During testing we apply 3 O-type functions that have never been applied to a trace trace with

---

Given an untested trace γ:
1) Determine all the canonical forms of γ
    1.1) COUNT γ as a trace to each equivalence space form.

2) FOR EACH canonical form from 1):
    2.1) COUNT all the O-type functions that have not yet been applied to a trace with that canonical form.

3) Determine all the inner traces of γ whose sub-instance classes are preserved in γ.

4) FOR EACH of the traces from 3):
    4.1) Determine all its canonical forms.
    4.2) FOR EACH canonical form:
        4.2.1) COUNT this inner trace as a path to that equivalence space IF this inner trace has not been previously counted.

5) Return the sum of the counts from steps 1.1), 2.1) and 4.2.1).

Figure A4.1: Algorithm for Counting PFS Elements

---

that canonical form. We have also tested DI as a trace to the same equivalence space. We do not count the "I" trace to that equivalence space as it is a mapping from outside our computation space and therefore technically not part of the PFS. From a practical standpoint it does not matter whether we count the "I" trace or not. Any trace of ADT functions must start with an I-type function, therefore the "I" trace would be exercised no matter what instance class we tested. Thus we have removed 1+3+1=5 elements from the PFS.

For the second trace tested there is one canonical form. We have tested the trace ADI that leads to the equivalence space of that canonical form. During testing we also apply 3 O-type functions that have never been applied to a trace with that canonical form. We do not count "DI" as it has already been counted in the first trace. Thus we have removed 3+1=4 elements from the PFS.

For the third trace tested there is one canonical form. We count the trace EEDDEI that leads to the equivalence space of the bag with no elements. We have already applied all O-type functions to a trace with this canonical form (first trace). We have also tested EDDEI, DDEI, DEI, and, EI as traces to the same equivalence space. Thus we have removed 1+1+1+1+1=5 elements from the PFS.

For the fourth trace tested there is one canonical form. We count the trace DEDEEI that leads to the equivalence space of the bag with no elements. We have already applied all O-type functions to a trace with this canonical form (first trace).

We have also tested EDEEI, DEEI, and, EEI as traces to the same equivalence space. We have have already tested "EI" (third trace) and cannot count it. Thus we have removed 1+1+1+1=4 elements from the PFS.

For the fifth trace tested there is one canonical form. We count the trace EDEDDI that leads to the equivalence space of the bag with no elements. We have already applied all O-type functions to a trace with this canonical form (first trace). We have also tested DEDDI, EDDI, and, DDI as traces to the same equivalence space. We have have already tested "DI" (first trace) and cannot count it. Thus we have removed 1+1+1+1=4 elements from the PFS.

For the sixth trace there are two canonical forms, one is bag with one element, the other is a bag with no elements. We have tested the trace "DDDEAI" to both equivalence spaces, thereby removing two elements from the PFS. Similarly we have also tested DDEAI, DEAI, and, EAI as traces to both equivalence spaces thereby removing 2+2+2= 6 more elements from the PFS. We have also tested AI as a trace to the single equivalence space of bags with one element. Thus we have removed 2+6+1=9 PFS elements.

For the seventh trace there is one canonical form. We count the trace AAAAI that leads to the equivalence space of bags with four elements. In testing this trace we apply 3 O-type functions that have never been applied to a trace with that canonical form. We also test AAAI as a trace to a bag of three elements and AAI as a trace

to a bag of two elements. "AI" has already been tested and removed from the PFS when the sixth trace was tested and therefore cannot be counted here. Therefore we remove 1+3+1+1=6 elements from the PFS.